Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Estudios de Postgrado
Maestría de Tecnologías de la Información y Comunicación

DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA EN AWS, QUE GARANTICE ALTA DISPONIBILIDAD DE UN SISTEMA WEB ORIENTADO A MICROSERVICIOS PARA LA TABULACIÓN Y PRESENTACIÓN DE LOS RESULTADOS DE LAS ELECCIONES DE GUATEMALA

Ing. César Aníbal Luis Alvarado
Asesorado por el Ing. Ms. Estuardo Zapeta

Guatemala, febrero de 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA EN AWS QUE GARANTICE ALTA DISPONIBILIDAD DE UN SISTEMA WEB ORIENTADO A MICROSERVICIOS PARA LA TABULACIÓN Y PRESENTACIÓN DE LOS RESULTADOS DE LAS ELECCIONES DE GUATEMALA

TRABAJO DE GRADUACIÓN

PRESENTADO A LA JUNTA DIRECTIVA DE LA
FACULTAD DE INGENIERÍA
POR

ING. CÉSAR ANÍBAL LUIS ALVARADO
ASESORADO POR EL ING. MS. ESTUARDO ZAPETA

AL CONFERÍRSELE EL TÍTULO DE

MAESTRO EN TECNOLOGÍAS DE LA INFORMACIÓN Y COMUNICACIÓN

GUATEMALA, FEBRERO DE 2022

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA

FACULTAD DE INGENIERÍA

NÓMINA DE JUNTA DIRECTIVA

DECANA	Inga. Aurelia Anabela Cordova Estrada
VOCAL I	Ing. José Francisco Gómez Rivera
VOCAL II	Ing. Mario Renato Escobedo Martínez
VOCAL III	Ing. José Milton de León Bran
VOCAL IV	Br. Kevin Vladimir Cruz Lorente
VOCAL V	Br. Fernando José Paz González
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

TRIBUNAL QUE PRACTICÓ EL EXAMEN GENERAL PRIVADO

DECANA	Inga. Aurelia Anabela Cordova Estrada
DIRECTOR	Mtro. Ing. Edgar Darío Álvarez Cotí
EXAMINADOR	Mtro. Ing. Marlon Antonio Pérez Türk
EXAMINADOR	Mtro. Ing. Estuardo Enrique Echeverría Nova
SECRETARIO	Ing. Hugo Humberto Rivera Pérez

HONORABLE TRIBUNAL EXAMINADOR

En cumplimiento con los preceptos que establece la ley de la Universidad de San Carlos de Guatemala, presento a su consideración mi trabajo de graduación titulado:

DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA EN AWS QUE GARANTICE ALTA DISPONIBILIDAD DE UN SISTEMA WEB ORIENTADO A MICROSERVICIOS PARA LA TABULACIÓN Y PRESENTACIÓN DE LOS RESULTADOS DE LAS ELECCIONES DE GUATEMALA

Tema que me fuera asignado por la Dirección de la Escuela de Estudios de Postgrado, con fecha noviembre de 2021.

Ing. César Aníbal Luis Alvarado

Decanato
Facultad de Ingeniería
24189101- 24189102
secretariadecanato@ingenieria.usac.edu.gt

LNG.DECANATO.OI.0113.2022

La Decana de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer la aprobación por parte del Director de la Escuela de Estudios de Posgrado, al Trabajo de Graduación titulado: **DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA EN AWS, QUE GARANTICE ALTA DISPONIBILIDAD DE UN SISTEMA WEB ORIENTADO A MICROSERVICIOS PARA LA TABULACIÓN Y PRESENTACIÓN DE LOS RESULTADOS DE LAS ELECCIONES DE GUATEMALA,** presentado por: **César Aníbal Luis Alvarado ,** que pertenece al programa de Maestría en artes en Tecnologías de la información y la comunicación después de haber culminado las revisiones previas bajo la responsabilidad de las instancias correspondientes, autoriza la impresión del mismo.

IMPRÍMASE:

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
DECANA
FACULTAD DE INGENIERÍA

Inga. Aurelia Anabela Cordova Estrada

Decana

Guatemala, febrero de 2022

AACE/gaoc

Guatemala, febrero de 2022

LNG.EEP.OI.113.2022

En mi calidad de Director de la Escuela de Estudios de Postgrado de la Facultad de Ingeniería de la Universidad de San Carlos de Guatemala, luego de conocer el dictamen del asesor, verificar la aprobación del Coordinador de Maestría y la aprobación del Área de Lingüística al trabajo de graduación titulado:

"DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA EN AWS, QUE GARANTICE ALTA DISPONIBILIDAD DE UN SISTEMA WEB ORIENTADO A MICROSERVICIOS PARA LA TABULACIÓN Y PRESENTACIÓN DE LOS RESULTADOS DE LAS ELECCIONES DE GUATEMALA"

presentado por **César Aníbal Luis Alvarado** correspondiente al programa de **Maestría en artes en Tecnologías de la información y la comunicación** ; apruebo y autorizo el mismo.

Atentamente,

"Id y Enseñad a Todos"

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA · FACULTAD DE INGENIERÍA · DIRECCIÓN · ESCUELA DE POSTGRADO

Mtro. Ing. Edgar Darío Álvarez Cotí
Director
Escuela de Estudios de Postgrado
Facultad de Ingeniería

Guatemala, 10 de julio 2021

M.A. Edgar Darío Álvarez Cotí
Director
Escuela de Estudios de Postgrado
Presente

M.A. Ingeniero Álvarez Cotí:

Por este medio informo que he revisado y aprobado el **TRABAJO DE GRADUACIÓN** titulado: "DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA EN AWS QUE GARANTICE UNA ALTA DISPONIBILIDAD DE UN SISTEMA WEB ORIENTADO A MICROSERVICIOS PARA LA TABULACIÓN Y PRESENTACIÓN DE LOS RESULTADOS DE LAS ELECCIONES DE GUATEMALA" del estudiante **César Aníbal Luis Alvarado** quien se identifica con número de carné **201990376** del programa de **Maestría en Tecnologías de la Información y la Comunicación.**

Con base en la evaluación realizada hago constar que he evaluado la calidad, validez, pertinencia y coherencia de los resultados obtenidos en el trabajo presentado y según lo establecido en el ***Normativo de Tesis y Trabajos de Graduación aprobado por Junta Directiva de la Facultad de Ingeniería Punto Sexto inciso 6.10 del Acta 04-2014 de sesión celebrada el 04 de febrero de 2014.*** Por lo cual el trabajo evaluado cuenta con mi aprobación.

Agradeciendo su atención y deseándole éxitos en sus actividades profesionales me suscribo.

Atentamente,

MARLON ANTONIO PEREZ TURK
INGENIERO EN CIENCIAS Y SISTEMAS
COLEGIADO No. 4492

MA. Ing. Marlon Antonio Pérez Türk
Coordinador
Maestría en Tecnologías de la Información y la Comunicación
Escuela de Estudios de Postgrado

Universidad de San Carlos de Guatemala
Facultad de Ingeniería
Escuela de Estudios de Postgrado
Coordinador de Área

Guatemala, 21 de octubre de 2021.

MSc. Ing. Edgar Álvarez Cotí
Director
Escuela de Estudios de Postgrados
Presente

Estimado MSc. Ing. Álvarez Cotí:

Reciba un cordial y atento saludo, a la vez aprovecho la oportunidad para hacerle de su conocimiento que he revisado y aprobado el trabajo de graduación titulado: "DISEÑO E IMPLEMENTACIÓN DE UNA ARQUITECTURA EN AWS QUE GARANTICE UNA ALTA DISPONIBILIDAD DE UN SISTEMA WEB QUE EMULA EL SISTEMA DEL TSE PARA LA TABULACIÓN Y PRESENTACIÓN DE LOS RESULTADOS DE LAS ELECCIONES DE GUATEMALA" del estudiante César Aníbal Luis Alvarado del Programa de Maestría en Tecnología de la Información y Comunicación, identificado con número de carné: *201990376*.

Agradeciendo su atención y deseándole éxitos en sus actividades profesionales me suscribo.

Atentamente,

"Id y enseñad a todos"

MSc. Ing. Edwin Estuardo Zapeta Gómez
Colegiado No. 12767
Asesor

Cc: Archivo/LA

Doctorado: Sostenibilidad y Cambio Climático. **Programas de Maestrías**: Ingeniería Vial, Gestión Industrial, Estructuras, Energía y Ambiente Ingeniería Geotécnica, Ingeniería para el Desarrollo Municipal, Tecnologías de la Información y la Comunicación, Ingeniería de Mantenimiento. **Especializaciones**: Gestión del Talento Humano, Mercados Eléctricos, Investigación Científica, Educación virtual para el nivel superior, Administración y Mantenimiento Hospitalario, Neuropsicología y Neurociencia aplicada a la Industria, Enseñanza de la Matemática en el nivel superior, Estadística, Seguros y ciencias actuariales, Sistemas de información Geográfica, Sistemas de gestión de calidad, Explotación Minera, Catastro.

ACTO QUE DEDICO A:

Mi madre	Que en paz descansa, por su amor y apoyo incondicional desde que comencé mis estudios básicos. Donde quiera que esté ahora, todos mis logros son para ella.
Padre Timoteo Conlan	Por haberme apoyado con una beca para realizar mis estudios del colegio y sus incontables consejos para triunfar en la vida.
Isabel Gutiérrez de Bosh	Que en paz descansa, por creer en mí y haberme otorgado la beca de la Fundación Juan Bautista Gutiérrez para realizar mis estudios universitarios, por su incansable apoyo y deseos de ánimos durante toda mi carrera.

AGRADECIMIENTOS A:

Universidad San Carlos de Guatemala	Por abrirme las puertas y haberme permitido adquirir nuevos conocimientos.
Mis catedráticos	Por su dedicación y por compartir sus conocimientos y experiencias en cada materia.
Mi asesor	Msc. Ing. Estuardo Zapeta, por su dedicación y constante apoyo para realizar este trabajo de investigación.
Servir, S. A.	Por permitirme utilizar un subdominio del dominio *servirsa.com* para realizar el proyecto de este trabajo de investigación.

ÍNDICE GENERAL

ÍNDICE DE ILUSTRACIONES V
LISTA DE SÍMBOLOS…… IX
GLOSARIO……. XI
RESUMEN…… XIX
PLANTEAMIENTO DEL PROBLEMA XXIII
OBJETIVOS……. XXVII
MARCO METODOLÓGICO XXIX
INTRODUCCIÓN……. XLV

1. ANTECEDENTES 1

2. JUSTIFICACIÓN……. 19

3. ALCANCES……. 23
   3.1. Perspectiva investigativa 23
   3.2. Perspectiva técnica 24
   3.3. Perspectiva de los resultados esperados 25

4. MARCO TEÓRICO 27
   4.1. Cloud Computing 27
      4.1.1. Nube pública 28
      4.1.2. Nube privada 28
      4.1.3. Nube híbrida 29
      4.1.4. Proveedor de servicios en la Nube 29
      4.1.5. Infraestructura como servicio 31

4.1.6. Plataforma como servicio 32
4.1.7. Software como servicio 33
4.1.8. Función como servicio 33
4.2. Amazon web services 35
4.2.1. Amazon EC2 37
4.2.2. Amazon Lambda – Serverless 38
4.3. Microservicios 40
4.3.1. Arquitectura de microservicios 41
4.3.2. REST API 45
4.4. Disponibilidad de las bases de datos 47
4.4.1. Escalar la lectura con una arquitectura Master-Slave 48
4.4.2. Escalar la escritura con una arquitectura sharding 49

5. PRESENTACIÓN DE RESULTADOS 51
5.1. Análisis y diseño de microservicios 51
5.2. Diseño de la arquitectura de alta disponibilidad en AWS 56
5.3. Diseño de la base de datos 59
5.4. Microservicios 62
5.5. Configuración de instancias de EC2 65
5.6. Configuración de AWS Lambda 68
5.7. Configuración de la BD en un RDBMS para una alta disponibilidad 72
5.7.1. Arquitectura de replicación Master-Slave 72
5.7.2. Configuración de la base de datos primaria (máster) 74
5.7.3. Configuración de las bases de datos secundarias (slave) 75
5.8. Aplicaciones web 79
5.8.1. Sistema de tabulación de resultados 79
5.8.2. Página de presentación de resultados 84

5.9. Rendimiento de la arquitectura de alta disponibilidad en AWS.... 89

6. DISCUSIÓN DE RESULTADOS 101
6.1. Análisis y diseño de microservicios 102
6.2. Modularización de las aplicaciones web 103
6.3. Disponibilidad a nivel de base de datos 104
6.4. Microservicios 106
6.5. Uso de AWS Lambda para una alta concurrencia 106
6.6. Rendimiento de instancias EC2 110
6.7. Optimización de MySQL para una alta concurrencia 111
6.8. Pruebas de carga 115
6.9. Oportunidades de mejora 118

CONCLUSIONES…….. 121

RECOMENDACIONES 123

REFERENCIAS……. 125

APÉNDICES……..... 133

ÍNDICE DE ILUSTRACIONES

FIGURAS

1. Configuración de la prueba de carga de la API de resultados..... XL
2. Prueba de Artillery de la API del sistema de tabulación - Parte 1 XLI
3. Prueba de Artillery de la API del sistema de tabulación - Parte 2 XLII
4. BD aplicación monolítica vs. BD aplicación orientada a microservicios . 7
5. Casos de uso del sistema Solteq Wellbeing..... 7
6. Arquitectura AWS Lambda de Solteq Wellbeing 9
7. API de Solteq Wellbeing en AWS Lambda..... 10
8. Arquitectura Lambda para la detección de actividades maliciosas 12
9. Modelo de replicación Master-Slave 17
10. Aplicación monolítica vs. microservicios vs. FaaS 34
11. Arquitectura de microservicios 42
12. Usando réplicas para escalar la lectura..... 49
13. Diagrama de modelo de dominio..... 52
14. Diagrama de casos de uso..... 53
15. Diagrama de módulos por casos de uso 54
16. Diagrama de microservicios 55
17. Arquitectura de alta disponibilidad en AWS..... 57
18. Diagrama de entidad-relación 60
19. API para el sistema de tabulación de resultados..... 63
20. API de consulta de la información pública..... 64
21. API de consulta de resultados..... 65
22. Descripción de una instancia de EC2..... 67
23. Grupo de seguridad de una instancia de EC2..... 68

24. Visualización de una Lambda en AWS 70
25. API Gateway como trigger para la Lambda de resultados 72
26. Arquitectura de replicación Master-Slave en un RDBMS 73
27. Configuración de una BD primaria en MySQL 74
28. Verificación del estado de la BD primaria 75
29. Configuración de una BD de réplica en MySQL 76
30. Verificación del estado de una BD de réplica 77
31. Configuración de una BD secundaria en MySQL para un alto rendimiento 78
32. Módulo de resultados del sistema de tabulación 81
33. Módulo de usuarios del sistema de tabulación 81
34. Módulo de departamentos y municipios del sistema de tabulación 82
35. Registro de mesa de votación en el sistema de tabulación 83
36. Formulario para el ingreso de resultados en el sistema de tabulación .. 84
37. Encabezado y filtros de la página de resultados 85
38. Gráfico de resultados tabulados 86
39. Resultados filtrados por la posición "alcalde" 87
40. Resumen de resultados tabulados 88
41. Tabla de mesas procesadas 88
42. Gráfico de abstencionismo 89
43. Estatus de replicación Master-Slave 92
44. Inicialización de pruebas de Artillery para ejecutar en paralelo 93
45. Estatus de las bases de datos antes y después de las pruebas 94
46. Lambda de resultados - resultados de Artillery 94
47. Lambda de resultados - distribución de latencia 95
48. Lambda de resultados - gráfico de disponibilidad 96
49. Lambda de resultados - métricas de la Lambda 97
50. Lambda de resultados - monitoreo de EC2 98
51. API del sistema de tabulación - resultados de Artillery 98

52. API del sistema de tabulación - distribución de latencia..... 99
53. API del sistema de tabulación - monitoreo de EC2 99
54. Consulta para obtener las mesas contabilizadas 119

TABLAS

I. Páginas del TSE alojados en un único servidor XXIV
II. Variables XXX
III. Configuración de la Lambda para la consulta de resultados 69
IV. Roles y permisos del sistema de tabulación..... 80
V. Configuración de la prueba final..... 90

LISTA DE SÍMBOLOS

Símbolo	Significado
B	*Byte*
EC2	*Elastic Compute Cloud*
I/O	*Input/Output*
IP	*Internet Protocol*
SSL / TLS	*Secure Socket Layer / Transport Layer Security*
vCPU	*Virtual CPU*

GLOSARIO

ALB	Application Load Balancer
API	Application programming interface
Asíncrona	Se dice que un proceso es asíncrono cuando no interrumpe la ejecución de un programa, es decir, permite que se ejecuten otros procesos mientras finaliza.
Assets	Terminología que se usa para hacer referencia a los recursos que una página web necesita para mostrarse de forma amigable a los usuarios: hojas de estilos, imágenes, fuentes, librerías de dependencia, entre otros.
AWS	Amazon Web Services
Backend	Término comúnmente usado en desarrollo de aplicaciones web para referirse a la lógica que se ejecuta en el lado del servidor.
BD	Base de Datos
Benchmark	Proceso que consiste en comparar dos productos o servicios para determinar las ventajas y desventajas de uno sobre otro.

Buffer	Memoria temporal que una aplicación o proceso utiliza para optimizar su rendimiento. La diferencia con la memoria RAM es que es de uso dedicado para la aplicación o proceso.
Caché	Memoria temporal que una aplicación o proceso utiliza para almacenar datos recurrentes para optimizar el acceso a ellos y su reutilización.
CLI	Command Line Interface
Cluster	Terminología que se usa en bases de datos distribuidas en el que dos o más computadoras están interconectadas para funcionar como un solo sistema.
CPU	Central Processing Unit
CRUD	Create – Read – Update – Delete
CSS	Cascading Style Sheets
CSV	Comma-Separated Values
Debugging	Proceso exhaustivo para encontrar un error en el código de un sistema.
Default	Valor por defecto de alguna configuración de alguna aplicación o servicio.
Deployment	En desarrollo web se refiere a las actividades que se deben realizar para llevar el código hasta

	producción, es decir, hacerlo disponible para el usuario final.
DNS	Domain name server
DoS	Denial of service
Downtime	Tiempo en que un sistema de información se queda inactivo.
Endpoint	En el desarrollo de sistemas web consiste en una URL desde donde se puede consumir una API.
Framework	Un marco de trabajo consiste en un conjunto de utilerías o componentes reutilizables que facilitan el desarrollo de las aplicaciones web.
Free Tier	Es un plan de AWS que permite explorar la mayoría de sus servicios sin ningún costo. Cada servicio tiene sus propias limitaciones.
Frontend	Término comúnmente usado en desarrollo de aplicaciones web para referirse a la lógica que se ejecuta del lado del cliente (navegador), es decir, la lógica de presentación.
GB	*Giga Byte*
Hardware	Componentes físicos y tangibles de una computadora o una arquitectura de red.

Header	En las solicitudes HTTP permiten que el cliente o el servidor adjunte información adicional a una solicitud, por ejemplo, el *token* de autenticación.
Hosting	Término que hace referencia al servicio de alojamiento en la nube para desplegar aplicaciones web, bases de datos o almacenamiento de archivos.
HTML	Hyper text markup language
HTTP	Hyper text transfer protocol
HTTPS	Hyper text transfer protocol secure
ID	Identificador único de registros en bases de datos relacionales.
InnoDB	Motor de procesamiento y almacenamiento por defecto de MySQL que garantiza las transacciones ACID. Vino a reemplazar MyISAM que es el motor original de MySQL en las versiones menores a 5.5.
JSON	JavaScript object notation
JWT	JSON Web Token
LB	Load balancer
Logs	Registros de bitácora de eventos.
Master (DB)	Base de Datos maestra o primaria en una arquitectura de replicación.

MB	Megabyte
NPM	Node package manager
On Premise	Nombre que se usa para las infraestructuras de tecnologías de la información que una empresa o entidad implementa y administra localmente sin depender de un proveedor como AWS o Microsoft Azure.
OSI	Open systems interconnection
Ping	Es un comando que se puede utilizar desde la línea de comandos para verificar si el servidor detrás de una IP o dominio se encuentra activo. También se puede utilizar para conocer a qué IP resuelve un dominio.
Postman	Aplicación de escritorio que permite probar rápidamente las APIs o endpoints de HTTP.
Query	Consulta de SQL.
RAM	Memoria volátil que una computadora utiliza para guardar la data de los programas y procesos que se están ejecutando.
RDBMS	Relational data base management system
REST	Representational state transfer

RESTful	Terminología que usa para indicar que una API sigue las convenciones REST.
Script	Archivo con instrucciones (código) en algún lenguaje de programación.
Serverless	Término utilizado en computación en la nube para hacer referencia a servicios donde la administración de los recursos y el software es responsabilidad del proveedor.
Slave (DB)	Base de Datos esclava o secundaria en una arquitectura de replicación.
SO	Sistema operativo
Software	Componentes lógicos y programas de computación que resuelven un problema. Normalmente consisten en miles de líneas de código escritos y compilados en algún lenguaje de programación.
SPA	Single page application
Streaming	Son datos que se generan continuamente a partir de varias fuentes y luego, ya sea se envían al mismo tiempo a otros objetivos en partes pequeñas o se espera que otros servicios la consuman.
TCP	Transmission control protocol
Thread	Un *thread* es un hilo de ejecución y se usa en programas que se pueden ejecutar en paralelo. Esta

	tecnología permite que una tarea grande se pueda dividir en subtareas más pequeñas y ejecutarlas en procesos separados para aprovechar las arquitecturas multi núcleo.
Timeout	Tiempo de respuesta agotado. Hace referencia al tiempo que un servicio o una aplicación espera antes de indicar que el recurso solicitado no está disponible.
UDP	User datagram protocol
VM	Virtual machine
VPC	Virtual private cloud

RESUMEN

Durante las elecciones presidenciales del año 2019 en Guatemala, hubo mucha confusión sobre los resultados que reportaba la página de resultados del Tribunal Supremo Electoral, ya que mostró incongruencias y por momentos se quedó en blanco o mostraban ceros en todos los resultados. Con el objetivo de investigar las posibles causas de estas anomalías y proponer una solución, se aventuró en este proyecto que propone una arquitectura en la nube de AWS que garantiza una alta disponibilidad para la consulta de los resultados de las elecciones y una arquitectura de microservicios para desacoplar el sistema de tabulación de la página de resultados.

Durante el planteamiento del problema se hizo una revisión de la página de resultados del TSE para explorar las características de la aplicación. Lo que se encontró es que la página seguía una arquitectura monolítica y compartía el servidor con otras aplicaciones del TSE. Rápidamente se comprendió que la página no seguía las buenas prácticas del desarrollo web moderno y tenía cierta dependencia con otros sistemas. Luego, pensando ya en una solución de prototipo, se planteó la siguiente pregunta: ¿Cómo diseñar e implementar una arquitectura en AWS que garantice una alta disponibilidad de un sistema web orientado a microservicios para la tabulación y presentación de los resultados de las elecciones de Guatemala, de tal manera que la arquitectura que sirve los resultados soporte una carga de 2000 solicitudes a cada 10 segundos sin afectar la disponibilidad del sistema de tabulación?

Una vez planteado el objetivo, se hizo una revisión de la literatura y los antecedentes para comprender los diferentes componentes de una arquitectura

de alta disponibilidad y los aspectos o situaciones que la pueden afectar. Finalizado este estudio se concluyó que, para aumentar la disponibilidad de la información a nivel de base de datos, se puede configurar una arquitectura de replicación *Master-Slave* en un RDBMS, de manera que la consulta de los resultados se realice sobre una BD secundaria y las operaciones de ingreso de resultados se realice únicamente sobre la BD primaria. Además, para manejar la alta concurrencia de solicitudes, se decidió implementar una arquitectura en AWS basándose principalmente en los servicios *Lambda function* y *EC2*.

Por otro lado, para desacoplar el sistema de tabulación de la página de resultados, se propuso una arquitectura de microservicios que consiste en 3 APIs autónomas que permiten la consulta y manipulación de la información. Cada microservicio cuenta con su propia Base de Datos que, aunque es la misma para las tres APIs, cada una opera de manera independiente y se aloja en su propia instancia de EC2. A nivel aplicativo, se desarrollaron dos aplicaciones web independientes como SPAs: el sistema de tabulación y la página de resultados. El primero consume una API RESTful y la segunda consume dos microservicios expuestos por dos funciones Lambdas que permiten consultar los resultados y la información pública para mostrarlos en gráficos amigables.

Finalmente, para validar la disponibilidad de la arquitectura se realizó una prueba de carga de 2000 solicitudes a cada 10 segundos durante 5 minutos hacia la Lambda de resultados y al mismo tiempo una prueba que valida la disponibilidad de la API del sistema de tabulación creando un registro de resultados a cada 10 segundos durante el mismo período de tiempo. Como resultado, en la prueba de la Lambda de resultados se obtuvo un nivel de disponibilidad del 99.95 % mientras que en la prueba de la API del sistema de tabulación se obtuvo un nivel de disponibilidad del 100 %, demostrando así la

independencia de los dos sistemas y la capacidad de la arquitectura para soportar una alta concurrencia de usuarios.

PLANTEAMIENTO DEL PROBLEMA

El proceso electoral de la primera vuelta de Guatemala del 2019 fue objetivo de muchas críticas, no solo por las sospechas de fraude que provocaron las diferentes anomalías que ocurrieron el día de la votación, sino también, por la irregularidad que mostró el sistema estadístico del Tribunal Supremo Electoral (TSE) al presentar los resultados preliminares del cotejo de votos. Tanto las redes sociales como los noticieros tienen evidencias de que el sistema dejó de responder durante y después de las elecciones, debido a la alta concurrencia de usuarios que consultaron la página de resultados preliminares, afectando tanto a los ciudadanos internautas como a los mismos administradores que necesitaban ingresar los resultados.

En un instante, la página de resultados mostró ceros en todos los totales, causando mucha incertidumbre y dudas sobre la fiabilidad del sistema informático del TSE (Sajquin, 2019). El mismo director de informática del TSE reconoció que hubo ciertas deficiencias en la recopilación de los datos, como lentitud y algunas fallas en el sistema de conteo (Morales, 2019). Además, aceptó que la página dejó de responder porque el número de consultas triplicó el número que ellos esperaban y con el que hicieron sus pruebas (Fáctica, 2019). Entre otros problemas de disponibilidad, se reportó el caso de que el sistema soportaba únicamente 20 candidatos, pero en algunas corporaciones municipales el número de candidatos era mayor, por lo que el sistema colapsó y terminó mostrando inconsistencias en los datos (Solano, 2019). Por último, luego que la mayoría de las mesas fueron digitalizadas, los fiscales políticos y los usuarios de internet notaron varias inconsistencias lo que obligó al TSE a revisar de nuevo las actas.

Durante esta revisión, el sistema volvió a fallar y tuvieron que suspender el conteo por varias horas hasta que este se reestableció (Álvarez, 2019).

Evidentemente el sistema mostró un problema de disponibilidad cuando una gran cantidad de usuarios accedieron al sitio al mismo tiempo. Esto se debe a un mal diseño tanto a nivel de infraestructura como a nivel aplicativo, aunque lo primero tiene más peso cuando se implementa un sistema de alta disponibilidad. La página mostraba tiempo de espera agotada (*timeout*) porque el servidor no era capaz de procesar todas las solicitudes y había un único punto de acceso, es decir, un solo servidor se encarga de recibir cada petición y devolver tanto el HTML de la página, como la data que se muestra en las gráficas. Este hecho se comprobó fácilmente revisando el dominio desde donde se descarga el HTML y el dominio al que se solicita la data. Las imágenes escaneadas de las actas se sirven también desde el mismo servidor. De hecho, todos los sitios del TSE para las elecciones estaban alojados en el mismo servidor. Para corroborar este hecho, se puede usar el comando *ping* desde una terminal para conocer la IP a la que apuntan los sitios.

Tabla I. **Páginas del TSE alojados en un único servidor**

Página	Dominio
Página oficial del TSE	ping tse.org.gt
Elecciones 2019	ping elecciones2019.tse.org.gt
Consulta de mesa de votación	ping endondevotas.tse.org.gt
Resultados preliminares	ping resultados2019.tse.org.gt

Fuente: elaboración propia.

Con esta configuración, un servidor tan potente como fuese no sería capaz de manejar tantas solicitudes al mismo tiempo. Lamentablemente no fue posible

realizar una prueba de estrés contra este servidor para determinar hasta cuántas sesiones puede soportar, ya que se podría ocasionar una denegación de servicio (DoS), lo cual es ilegal y penado por la ley.

Las constantes caídas del sistema afectaron a los usuarios que únicamente querían conocer los resultados, pero también afectó a los operadores que necesitaban ingresar la información y, en consecuencia, atrasó al TSE para la entrega de los resultados definitivos. También cabe destacar que esto generó mucha desconfianza e incertidumbre en la población en general y en los fiscales de los partidos políticos, dando lugar a acusaciones de fraude o posible manipulación de la información.

Se trata de un sistema crítico que maneja información sensible y de mucho valor, por lo que se debe garantizar una continuidad operacional y una alta disponibilidad, o al menos mientras no sea por un problema de red o de energía. En el peor de los casos, si llegara a quedar fuera de servicio por cualquier causa no contemplada, el tiempo de baja (*downtime*) debería ser lo menor posible de manera que no perjudique tanto el desarrollo de las operaciones como a los usuarios que consultan los resultados.

Para tener un valor aproximado del número de usuarios concurrentes que debe soportar el sitio de resultados del TSE, se hizo un análisis simple que se basa en el número total de visitantes y el tiempo promedio que dura una visita. De acuerdo con un análisis al sitio *resultados2019.tse.org.gt* realizado con la herramienta web SimilarWeb (*https://www.similarweb.com*), se encontró que durante el mes de junio hubo 154,366 visitantes únicos y, además, el tiempo promedio de una visita fue de 12 minutos. El artículo web *Estimando usuarios concurrentes* de EVERAC99, propone la siguiente fórmula para dar una estimación del número promedio de usuarios concurrentes: $C = n * L / T$, donde

n es el número de total de usuarios, L el tiempo promedio de una visita y T el intervalo de tiempo a analizar (EVERAC99, 2009). Si analizamos 16 horas (960 mins) de disponibilidad durante un día, tenemos que el número posible de usuarios concurrentes es de $C = (154{,}366 * 12\ mins)/(960\ mins) = 1{,}930$.

Planteado este problema tecnológico, se formula la siguiente pregunta de investigación: ¿Cómo diseñar e implementar una arquitectura en AWS que garantice una alta disponibilidad de un sistema web orientado a microservicios para la tabulación y presentación de los resultados de las elecciones de Guatemala, de tal manera que la arquitectura que sirve los resultados soporte una carga de 2000 solicitudes a cada 10 segundos sin afectar la disponibilidad del sistema de tabulación?

Y para llegar a la solución, se plantean las siguientes preguntas auxiliares:

- ¿Cómo configurar en un RDBMS una base de datos con redundancia de tipo Master-Slave para garantizar una alta disponibilidad de la lectura de los resultados?
- ¿Cómo modularizar los sistemas de tabulación y presentación de resultados para que puedan operar de manera independiente a nivel aplicativo, siguiendo un patrón de microservicios?
- ¿Cómo configurar una arquitectura de alta disponibilidad en AWS usando los servicios de AWS Lambda y Elastic Compute Cloud (EC2)?
- ¿De qué manera se puede verificar que efectivamente la página de resultados soporta hasta 2000 solicitudes concurrentes a cada 10 segundos sin afectar la operatividad del sistema de tabulación de votos?

OBJETIVOS

General

Diseñar e implementar una arquitectura en AWS que garantice alta disponibilidad de un sistema web orientado a microservicios para la tabulación y presentación de los resultados de las elecciones de Guatemala, de tal manera que la arquitectura que sirve los resultados soporte hasta 2000 usuarios concurrentes con un nivel de disponibilidad del 99.9 % y sin afectar la disponibilidad del sistema de tabulación.

Específicos

- Configurar en un RDBMS una base de datos con redundancia de tipo Master-Slave para garantizar una alta disponibilidad de la lectura de los resultados.

- Modularizar los sistemas de tabulación y presentación de resultados para que puedan operar de manera independiente a nivel aplicativo, siguiendo un patrón de microservicios.

- Configurar una arquitectura en AWS que garantice un nivel de disponibilidad del 99.9 % para una carga de 2000 usuarios concurrentes, usando los servicios de Lambda y Elastic Compute Cloud (EC2).

- Realizar una prueba de carga de 2000 sesiones concurrentes solicitando los resultados a cada 10 segundos durante un período de 5 minutos y

demostrar la disponibilidad del sistema de tabulación mediante otro proceso paralelo que manda a guardar un registro de resultados a cada 10 segundos durante el mismo período de tiempo.

MARCO METODOLÓGICO

Tipo de investigación

La metodología de estudio aplicada en este proyecto de investigación es mixta ya que para identificar el problema se hizo un análisis subjetivo de los hechos (análisis cualitativo) y luego se propone una solución medible mediante pruebas experimentales (análisis cuantitativo). Es cualitativa porque se analizaron los hechos del problema de disponibilidad de la página de resultados del TSE y se hicieron inferencias acerca de las posibles causas. Luego, con base en estas deducciones se plantea una arquitectura prototipo en la nube de Amazon que se espera garantice una alta disponibilidad. Debido a que la disponibilidad de un sistema no depende únicamente de la infraestructura, también se propone una arquitectura de microservicios para independizar la página de resultados del sistema de tabulación de votos. El estudio es también cuantitativo porque se realizaron experimentos (pruebas) para determinar la configuración óptima de MySQL y la instancia de EC2 para la base de datos de resultados que debe soportar una alta concurrencia.

Diseño de investigación

La investigación utiliza un diseño experimental para presentar los resultados y encontrar la solución óptima. Esto debido a que se necesitó realizar varias pruebas de carga para determinar las configuraciones óptimas de las Lambdas, la instancia de MySQL y las instancias de EC2 de las Bases de Datos secundarias. Se analizó el resultado de cada iteración de pruebas y cuando se

logró satisfacer el objetivo principal, se usó la configuración obtenida para realizar las pruebas integradas de la arquitectura completa.

Alcances

El presente trabajo de investigación posee un enfoque explicativo y descriptivo. Durante la identificación del problema posee un enfoque explicativo ya que se aclaran los hechos y se intenta demostrar las razones por las que las páginas del TSE quedaron fuera de servicio durante las elecciones de Guatemala de 2019. Luego adquiere un enfoque descriptivo porque propone una solución en la nube que garantiza una alta disponibilidad y describe los componentes y patrones de diseño que se aplicaron para alcanzar los objetivos planteados.

Variables

Debido a que la investigación posee un diseño experimental, durante las pruebas realizadas se identificaron las variables siguientes para validar si se alcanzaron los objetivos total o parcialmente.

Tabla II. **Variables**

Variables	Definición	Subvariables	Indicadores
Disponibilidad	Capacidad de la arquitectura en soportar una alta carga de usuarios en la página de resultados.	• Solicitudes exitosas • Solicitudes fallidas	Porcentaje de nivel de disponibilidad de la arquitectura que sirve los resultados.

Continuación tabla II.

Concurrencia reservada de las Lambdas	Valor de la "concurrencia reservada" de las Lambdas para soportar una carga de 2000 solicitudes a cada 10 segundos.	• Número de errores en la Lambda por *timeout*. • Cantidad de solicitudes rechazadas.
Cantidad de vCPUs	Cantidad de vCPUs para la instancia de EC2 de una BD secundaria para soportar una alta concurrencia.	• Porcentaje de uso del procesador.
Cantidad de RAM	Cantidad de RAM para la instancia de EC2 de una BD secundaria para soportar una alta concurrencia.	• Porcentaje de uso de la RAM. • Número de errores de conexión.
Latencia	Tiempo que demora una solicitud de consulta de resultados en resolverse.	• Tiempo mínimo • Tiempo máximo • Tiempo promedio • p99 (percentil 99) • p95 (percentil 95)

Fuente: elaboración propia.

Técnicas de recolección de información

Durante la etapa de identificación del problema se utilizó la técnica de observación experimental ya que se hicieron suposiciones sobre las razones por las que la página de resultados del TSE quedó inactiva, se hicieron algunos experimentos y se utilizó una ficha de registro para anotar las observaciones. También se utilizó la técnica de revisión documental para investigar sobre las noticias en los periódicos electrónicos acerca de los problemas de disponibilidad que sufrieron los sistemas del TSE, reportados durante el día de la votación y durante el proceso de tabulación de los votos.

Durante la recopilación de la información acerca de las tecnologías que se utilizaron, se aplicaron las técnicas de revisión documental y revisión analítica. Luego en la fase de diseño se utilizó el método de Diagramas de Flujos para plasmar en papel los flujos de comunicación y la integración de los componentes de la arquitectura.

La evaluación y validación de la arquitectura en AWS requiere de un proceso experimental por lo que la técnica apropiada para la recopilación de los datos cuantitativos es la tabulación de datos mediante fichas de registro y generación de gráficos del porcentaje de disponibilidad. Los resultados de cada iteración de pruebas se anotaron en las fichas de registro, lo cual incluye los resultados generados por la herramienta que se utilizó para realizar las pruebas de carga y las métricas de Amazon CloudWatch sobre el rendimiento de las Lambdas y las instancias de EC2.

Fases

Fase I. Identificación del problema

La primera fase del proyecto consistió en la identificación del problema y el análisis de sus posibles causas. Durante las elecciones generales del año 2019, varios medios de comunicación y el público en general, reportaron problemas de disponibilidad tanto de la página para consultar los resultados como del sistema para la tabulación de los votos. Luego de una revisión documental de los reportajes y publicaciones de los periódicos electrónicos, donde se discutían las causas del inconveniente como también las declaraciones de los responsables de dicho sistema, se procedió a estudiar las páginas del TSE para intentar justificar las causas del problema.

El primer acercamiento fue el análisis de las IPs a las que resuelven las distintas páginas del TSE y lo que se descubrió fue que la mayoría de las páginas del TSE estaban alojadas en un único servidor. Esto definitivamente afecta la disponibilidad de los sistemas. Luego, también se notó que las páginas seguían una arquitectura monolítica ya que todo el contenido (datos y *assets*) provenía del mismo servidor. En las aplicaciones web modernas, se hace la distinción entre *frontend* y *backend* para separar la vista de la lógica para manipular los datos.

Luego de analizar el problema, se concluyó que evidentemente se debía a un mal diseño de la arquitectura en la nube y la alta cohesión que existía entre los sistemas. Con el objetivo de proponer una solución a dicho problema, se plantea un sistema que se compone de pequeños microservicios, cada uno con una función específica e independiente de los demás para reducir el nivel de acoplamiento. Como prototipo se consideró únicamente la página de resultados y la aplicación para la tabulación de los votos. Por otro lado, un sistema orientado

a microservicios requiere también de una infraestructura que se ajuste a este patrón de diseño y ayude a alcanzar los mismos objetivos. Para ello, se decidió investigar acerca de cómo construir arquitecturas altamente disponibles en AWS y cómo integrarlos con microservicios.

Fase II. Revisión documental

La siguiente fase del proyecto consistió en una investigación profunda acerca de los distintos factores que afectan la disponibilidad de un sistema de información. Luego de identificar estos componentes, se procedió con la revisión documental de cada uno consultando libros, publicaciones, tesis, artículos científicos y páginas de internet para entender con toda claridad en qué consiste, cómo afecta a la disponibilidad, qué tecnologías disponibles existen, qué aspectos deben tomarse en cuenta para elegir una tecnología u otra y cómo integrar todos estos bloques.

De la misma manera, se estudiaron varios de los servicios en la nube de Amazon para determinar cuáles son apropiados para el proyecto y cuáles se ajustan a una arquitectura de microservicios. Esta fase provee el marco teórico para implementar los distintos componentes de la arquitectura.

Fase III. Familiarización con AWS

Para implementar la arquitectura en la nube de Amazon, fue necesario primero familiarizarse con cada servicio por utilizar y saber cómo integrarlos para que funcionen como un sistema completo. Para ello, se tomó y completó el curso *AWS Certified Solutions Architect - Associate 2019* de la plataforma Udemy, el cuál provee los conocimientos necesarios para adquirir la experiencia y convertirse en un arquitecto certificado de AWS.

Fase IV. Diseño de la arquitectura

Esta fase consistió en el diseño en papel de la arquitectura, tanto a nivel aplicativo, como a nivel de infraestructura. Específicamente, consistió en la definición de los microservicios y la arquitectura de integración de los servicios de AWS. Como resultado de esta fase se obtuvo el esquema de la solución en la nube y el esquema de comunicación entre aplicaciones y microservicios.

En esta fase se utilizó la técnica de recolección de información mediante diagramas de flujos, la cual consiste en mostrar de manera gráfica el flujo de la información y la comunicación entre los componentes de un sistema.

Los diagramas que se realizaron sobre la arquitectura lógica son:

- Diagrama de modelo de dominio
- Diagrama de casos de uso
- Diagrama de módulos por casos de uso
- Diagrama de microservicios
- Diagrama de entidad-relación

Los diagramas que se realizaron sobre la arquitectura física son:

- Arquitectura de alta disponibilidad en AWS
- Diagrama de replicación Master-Slave

Fase V. Implementación de microservicios

Esta fase comprende el desarrollo de las APIs en forma de microservicios para la consulta de los resultados y el sistema de tabulación. El microservicio del sistema de tabulación es una API RESTful que permite el ingreso de los registros de resultados y la creación de registros de las otras tablas que complementan el sistema. Además, maneja las peticiones relacionadas a la creación de usuarios y autenticación, ya que las llamadas para ingresar los resultados deben estar autenticadas. Esta API se implementó en el lenguaje NodeJS utilizando el *framework express* y se utilizó el ORM de NodeJS llamado Sequelize para facilitar la consulta y manipulación de los datos.

Por otro lado, para la página de consulta de los resultados se implementaron dos microservicios que consisten en dos funciones de NodeJS que se pueden ejecutar en una Lambda de AWS. El primero permite consultar la información pública como los departamentos, los países, los partidos políticos, entre otros. El segundo permite consultar específicamente los registros de resultados y los candidatos que aplican para los filtros que se seleccionen desde la página web. Para probar las funciones de las Lambdas localmente antes de desplegarlos en AWS Lambda, se implementó una versión local que se expone en un servidor local de NodeJS y se conecta a una BD local.

Fase VI. Configuración de la arquitectura de replicación *Master-Slave*

En esta fase se realizó una revisión de la literatura para comprender cómo funciona la arquitectura *Master-Slave* en los RDBMS y qué ventajas ofrece. Luego se realizó una prueba de concepto de esta arquitectura con dos instancias de EC2 usando el RDBMS de MySQL. Se procedió a crear dos instancias en la consola de EC2 de tipo “*t2.micro*” y se instaló el servidor de MySQL.

Se realizaron las configuraciones mínimas de las BDs primaria y secundaria para habilitar la conexión entre ambas e iniciar el proceso de sincronización. Se creó una BD y una tabla de pruebas y luego se comprobó que la sincronización se realiza correctamente cada vez que se ingresa, actualiza o elimina un registro de la tabla en la BD primaria. Con esta prueba de concepto realizada, se tuvo la seguridad de que la arquitectura funciona y se podía continuar con el proyecto.

Fase VII. Implementación de aplicaciones prototipos

En esta fase se desarrollaron las 2 aplicaciones web como *Single Page Application (SPA)*: página de consulta de resultados y página de tabulación de resultados. Estas aplicaciones fueron implementadas con el lenguaje angular 11 y se usó la librería de estilos Bootstrap CSS para estilizar las interfaces.

Fase VIII. Configuración y despliegue de la arquitectura

Esta fase consistió en la configuración e integración de los diferentes servicios que se usaron de AWS. La API del sistema de tabulación se desplegó en una instancia de EC2 de tipo *t2.micro* usando un servidor de *nginx*. Además, se configuró con un nombre de dominio que permitió utilizar un certificado de seguridad de *Let's Encrypt* generado con la herramienta *certbot*. De esta manera, la data que se intercambia entre la API y la aplicación web del sistema de tabulación es encriptada bajo el protocolo de SSL/TLS.

Una vez configurada la API del sistema de tabulación, se procedió a configurar y habilitar la arquitectura *Master-Slave* de las bases de datos, pero ya con la BD real que posee todas las tablas de la aplicación. Además, dentro de la API se desarrollaron varios procesos de carga de datos iniciales para no tener

que crearlos manualmente, que en Sequelize se llaman *seeders*. Se crearon *seeders* para todas las tablas del sistema con el objetivo de automatizar el proceso de despliegue (*deployment*) de la API y las BDs. Incluso, se crearon registros de resultados de forma aleatoria para tener datos listos para las pruebas de carga.

Una vez que el *cluster* de bases de datos se encontraba operando, se configuraron las funciones de AWS Lambda y se cargó el código de cada función junto con sus dependencias. Luego se registraron las variables de entorno que permiten la conexión hacia las BDs secundarias y se procedió a realizar una prueba desde la consola de AWS Lambda para verificar que la Lambda puede comunicarse con su BD correspondiente. Una vez se logró establecer esta comunicación, se habilitó para cada Lambda un *endpoint* HTTP público usando el servicio API Gateway de AWS.

Por último, se trabajó el despliegue de las aplicaciones web en la nube de Amazon y para ello se utilizó el servicio de AWS Amplify, que permite levantar y desplegar SPAs en cuestión de minutos. Una de las ventajas que se aprovechó de este servicio es la automatización de la publicación de cambios, ya que gracias a su integración con Github, cada vez que se manda un cambio al repositorio, Amplify se encarga de obtener automáticamente estos cambios y distribuir la nueva versión de la página a través de la red de Amazon CloudFront.

Fase IX. Pruebas de carga

Una vez las aplicaciones y los microservicios fueron desplegados en los diferentes servicios de AWS, se procedió con la tarea más importante, la cual consiste en explotar las capacidades de la arquitectura para averiguar la carga máxima que podía soportar y realizar los ajustes necesarios para lograr el objetivo de soportar 2000 solicitudes de consulta de resultados a cada 10 segundos sin afectar la disponibilidad del sistema de tabulación.

Las pruebas de carga se realizaron contra la Lambda que devuelve los registros de resultados, es decir, la API que se consume desde la página de resultados, ya que esta es la que va a recibir una alta carga de solicitudes. Una vez se encontró la configuración óptima para alcanzar un nivel de disponibilidad de al menos del 99.9 %, se realizó una prueba de integración que involucra también a la API del sistema de tabulación con el objetivo de demostrar la disponibilidad de este sistema independiente de la disponibilidad de la página de resultados.

Para ejecutar las pruebas de carga se utilizó la herramienta Artillery distribuida por NPM. Artillery es una librería de NodeJS de código abierto (aunque existe en versión pagada) que permite realizar pruebas de estrés de APIs transaccionales y cualquier servicio que se pueda comunicar a través de una red (Artillery Software Inc., 2021). Esta herramienta permitió configurar el escenario de enviar 2000 solicitudes a cada 10 segundos a la API de resultados durante cierto periodo de tiempo y obtener las métricas sobre el rendimiento de la arquitectura como la cantidad de solicitudes exitosas, el número de solicitudes que fallaron, el número de errores por código de estatus HTTP y los tiempos de respuesta.

Las pruebas de Artillery se definen en un archivo YAML, donde se especifica el *endpoint* a consumir, los escenarios y las fases de la prueba. Los escenarios definen el flujo de acciones que un usuario real en el sistema realizaría y las fases definen la cantidad de usuarios que se esperan durante cierto periodo de tiempo, la cual puede crecer linealmente o tener picos a cada cierto tiempo. La figura 1 muestra la configuración que se utilizó para probar la API de resultados, aunque solo muestra un periodo de 30 segundos y una cantidad de 100 usuarios nuevos a cada 10 segundos. En el apéndice 6 se muestra la configuración completa de la prueba.

Figura 1. **Configuración de la prueba de carga de la API de resultados**

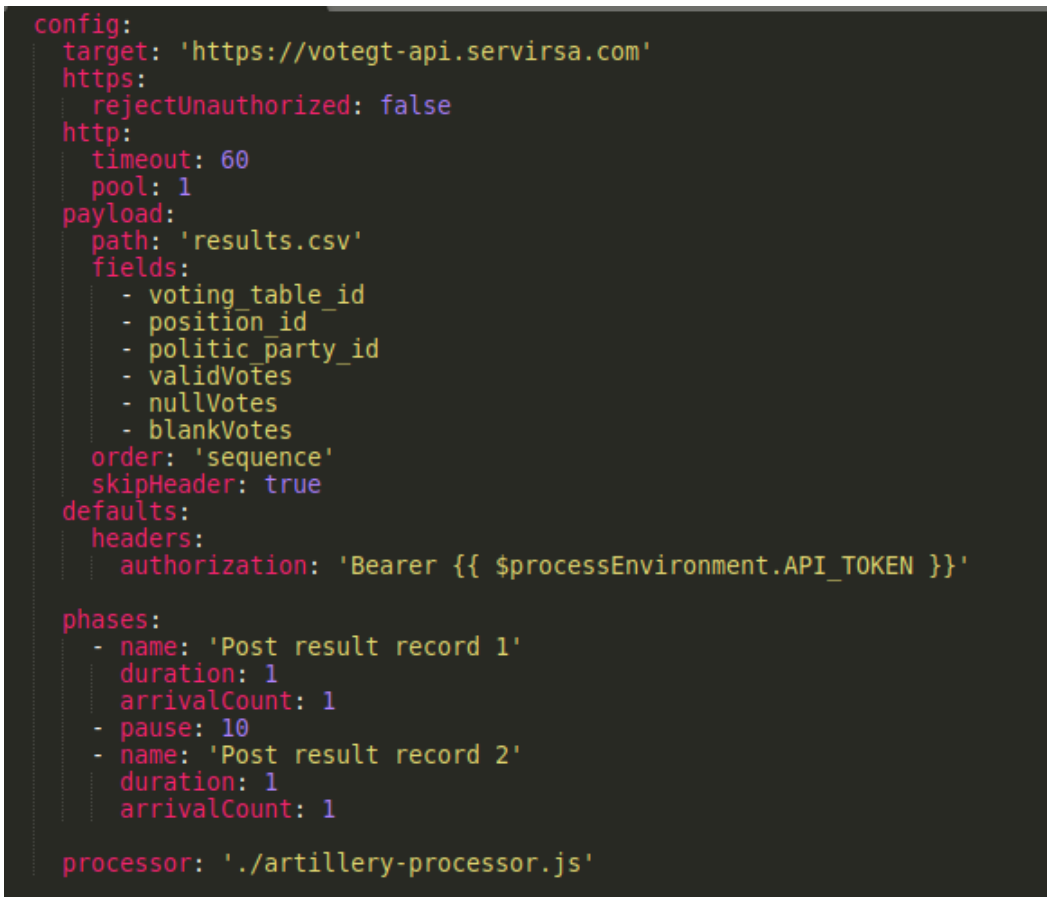
```
config:
  target: 'https://lmjaptb3tc.execute-api.us-east-1.amazonaws.com'
  http:
    timeout: 60

  phases:
    - duration: 1
      arrivalCount: 100
      name: 'Get results 1'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 2'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 3'

scenarios:
  - name: 'Get vote results'
    flow:
      - get:
          url: '/v1/results?positionId=1'
```

Fuente: elaboración propia, utilizando Sublime Text.

Por otro lado, para probar la API del sistema de tabulación se configuró una prueba de Artillery que manda a crear un registro de resultado a cada 10 segundos durante 5 minutos. Esta prueba se ejecutó en paralelo con la prueba

de la Lambda de resultados para demostrar la independencia de los dos sistemas. En la Figura 2 se muestra la primera parte del archivo de configuración de esta prueba y para fines de demostración, esta prueba solo genera dos solicitudes en 10 segundos. Las solicitudes a la API del sistema de tabulación deben ir autenticadas y es por ello por lo que el archivo de configuración obtiene el *token* (JWT) de autenticación de una variable de entorno y lo agrega en el *header* de autorización.

Figura 2. **Prueba de Artillery de la API del sistema de tabulación - Parte 1**



```yaml
config:
  target: 'https://votegt-api.servirsa.com'
  https:
    rejectUnauthorized: false
  http:
    timeout: 60
    pool: 1
  payload:
    path: 'results.csv'
    fields:
      - voting_table_id
      - position_id
      - politic_party_id
      - validVotes
      - nullVotes
      - blankVotes
    order: 'sequence'
    skipHeader: true
  defaults:
    headers:
      authorization: 'Bearer {{ $processEnvironment.API_TOKEN }}'

  phases:
    - name: 'Post result record 1'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 2'
      duration: 1
      arrivalCount: 1

  processor: './artillery-processor.js'
```

Fuente: elaboración propia, utilizando Sublime Text.

Los resultados que se envían en las solicitudes se cargaron desde el archivo “results.csv”, el cual fue generado desde la API del sistema de tabulación y contiene registros válidos. El escenario de la prueba consiste en generar una solicitud con el verbo POST como se muestra en la Figura 3, enviando como data la mesa de votación, la posición, el partido político y la cantidad votos válidos, nulos y en blanco. Al finalizar, se imprime la respuesta de la API mediante la función *printStatus* que está definida en el *script* que se definió como *processor* en la Figura 2. La configuración completa de la prueba se presenta en el apéndice 7.

Figura 3. **Prueba de Artillery de la API del sistema de tabulación - Parte 2**

```
scenarios:
  - name: 'Post result record'
    flow:
      - log: 'POST data: { voting_table_id: {{ voting_table_id }},
      - post:
          url: '/results'
          json:
            voting_table_id: '{{ voting_table_id }}'
            position_id: '{{ position_id }}'
            politic_party_id: '{{ politic_party_id }}'
            validVotes: '{{ validVotes }}'
            nullVotes: '{{ nullVotes }}'
            blankVotes: '{{ blankVotes }}'
          afterResponse: 'printStatus'
```

Fuente: elaboración propia, utilizando Sublime Text.

La herramienta de Artillery permite guardar los resultados de una iteración de pruebas en un archivo JSON y luego con la misma herramienta, se generó un reporte más amigable en formato HTML. De este reporte se obtuvo el número de éxitos y el número de errores, y con estos valores se calculó el porcentaje de disponibilidad de la arquitectura.

Además de la salida de Artillery, también se aprovecharon los gráficos de monitoreo que ofrece AWS con su servicio Amazon CloudWatch para tener una mejor comprensión sobre el rendimiento de los componentes de la arquitectura. Se configuró un *dashboard* en Amazon CloudWatch que permitió visualizar en una sola pantalla los gráficos que interesan tanto de la Lambda como de la instancia de la base de datos.

De la Lambda se analizaron las siguientes métricas: número de invocaciones, tiempo de duración de las solicitudes, número de errores, porcentaje de éxito, número de ejecuciones concurrentes y cantidad de solicitudes rechazadas por sobre pasar el límite de concurrencia reservada. De la instancia de EC2 se graficó el porcentaje de uso del procesador y el tráfico de entrada y salida de la red.

Artillery tiene la capacidad de ejecutar las pruebas en contenedores de ECS (*Elastic Container Service*), el cuál es un servicio de Amazon que permite ejecutar tareas en un *cluster* de contenedores de forma sincronizada, ya sea administrados por uno mismo (instancias de EC2) o la opción *serverless* que ofrece Fargate. AWS Fargate elimina la necesidad de provisionar y administrar servidores, permite especificar los recursos por cada contenedor y el cobro es únicamente por el tiempo de uso y el tráfico de red generado (Amazon Web Services, Inc., 2021).

Para provisionar el *cluster* de ECS Fargate, Artillery dispone de un comando que utiliza el CLI de AWS para crear todos los recursos que requiere el *cluster* para ejecutar las pruebas. En el Apéndice 8 se documenta el proceso para inicializar el *cluster* de ECS utilizando Fargate y se muestra el comando que se utilizó para ejecutar las pruebas y generar el reporte en formato HTML.

Fase X. Análisis y presentación de resultados

Por último, se realizó el análisis y discusión de los resultados encontrados, las observaciones y las lecciones aprendidas. También se descubre si se alcanzaron los objetivos completa o parcialmente, se explica las fortalezas y deficiencias de la arquitectura, específicamente, sobre la capacidad de soportar una concurrencia de 2000 solicitudes de consultas de resultados a cada 10 segundos.

En esta fase también se trabajó toda la documentación relacionada con la configuración de los servicios en AWS y la configuración de la arquitectura de replicación *Master-Slave*.

INTRODUCCIÓN

Las elecciones de primera vuelta del año 2019 para elegir a las nuevas autoridades que dirigirían el país de Guatemala en los siguientes 4 años, estuvo lleno de controversias debido a los problemas de disponibilidad que mostraron los sistemas del Tribunal Supremo Electoral (TSE), específicamente el sistema de tabulación de votos y la página para consultar los resultados preliminares. Esta deficiencia generó muchas dudas en la población y desató las críticas de los medios de comunicación. Los encargados del Departamento de Informática del TSE admitieron que el elevado número de usuarios que consultaron la página de resultados colapsó su arquitectura y dejó fuera de servicio el sistema para tabular los votos.

A raíz de este problema, surge la idea de este proyecto para plantear una solución que haga frente a este tipo de exigencias y garantice una alta disponibilidad y confiabilidad. Hoy en día, los sistemas de información que soportan los procesos de negocio de grandes empresas de escala mundial deben estar preparados para soportar la alta concurrencia de usuarios. Plataformas como Netflix, Amazon E-Commerce, Spotify, Facebook, YouTube, entre otros, responden a millones de solicitudes por segundo y rara vez presentan problemas de disponibilidad. Entonces surge la pregunta de ¿por qué el sistema del TSE no soportó unos cuántos miles de usuarios consultando la página de resultados? La respuesta es que la arquitectura lógica y física de dicho sistema no estaba diseñada para soportar una alta concurrencia.

Este proyecto justamente propone una arquitectura lógica y física que garantiza la continuidad y disponibilidad de un sistema de información, incluso

cuando hay una alta carga de usuarios. Específicamente, se propone e implementa una arquitectura prototipo en la nube de AWS que sea capaz de manejar una concurrencia de 2000 solicitudes de consulta de resultados a cada 10 segundos. La arquitectura física se basa en un sistema de bases de datos con redundancia de tipo *Master-Slave* para disponer de varias fuentes de información y explota las características de los servicios de AWS Lambda y EC2 para soportar una alta concurrencia.

Se desarrollaron 2 prototipos de las aplicaciones que utiliza el TSE para la gestión de los resultados de las elecciones: el sistema de tabulación de votos y la página de consulta de resultados. Ambas aplicaciones se implementan como SPAs para separar la lógica de presentación de la lógica de manipulación de los datos. La arquitectura lógica se basa en el paradigma de microservicios para disponer de tres APIs totalmente desacopladas y cada una sirve para un propósito específico. El sistema de tabulación consume una API RESTful autenticada, mientras la página de resultados consume un *endpoint* para consultar la información pública como los departamentos, municipios, partidos políticos, entre otros, y otro *endpoint* para consultar específicamente los registros de resultados.

La API del sistema de tabulación se despliega en una instancia de EC2 y se conecta a la BD primaria alojada en otra instancia de EC2. Cada API pública se despliega en una función de AWS Lambda y se conecta a una BD secundaria. Cada BD secundaria se aloja en una instancia de EC2 optimizada para soportar una alta carga de consultas de lectura.

Luego para evaluar el rendimiento de la arquitectura, se realizó una prueba de carga hacia la Lambda de resultados de 2000 solicitudes a cada 10 segundos durante 5 minutos y al mismo tiempo, se ejecutó otra prueba que crea un registro

de resultado en la API del sistema de tabulación y con ello demostrar la independencia de los dos sistemas implementados.

1. ANTECEDENTES

Existen varios factores que mejoran o perjudican a un sistema altamente disponible, desde su implementación en *software* hasta la infraestructura donde se despliega. A continuación se presenta una investigación sobre casos de uso de los componentes más importantes de una arquitectura que ofrece una alta disponibilidad, empezando con la infraestructura en la nube y sus garantías, luego se hace una valoración del uso de contenedores *docker* para la implementación de un sistema resiliente, se presenta también una investigación que presenta un caso de uso donde se expone la mayoría de los servicios de AWS que se van a utilizar en este proyecto, se discute en un artículo las ventajas y limitaciones de la tecnología *serverless* y por último se presenta un estudio sobre cómo puede mejorarse la disponibilidad desde la capa de base de datos.

El artículo que lleva por nombre *Load balancing in cloud Computing: A big picture*, trabajo de Sambit Mishra, Bibhudatta Sahoo y Priti Parida, en el año 2018, presenta un panorama general de la complejidad que tiene el problema del balanceo de carga en la nube. Normalmente, el balanceo de carga en la nube con un sistema de objetivos múltiples es bien conocido como un problema de completitud NP. Los investigadores han presentado diferentes técnicas heurísticas para lograr una solución subóptima para el balanceo de cargas en ambientes de la nube (Mishra *et al*., 2017). La complejidad de esta tarea viene dada por diferentes factores que se deben considerar para lograr un alto rendimiento, tales como la estrategia de aplicación (estática o dinámica), la

métrica para decidir a qué máquina asignar una tarea, la capa del modelo OSI dónde se quiere intervenir, entre otros.

Esta complejidad crece aún más cuando se mezcla el balanceo de carga con la tolerancia a fallos porque en este tipo de arquitectura se realizan migraciones de tareas o incluso, migraciones de máquinas virtuales enteras. Si por cualquier razón una tarea falla durante su ejecución en una máquina virtual (VM), se traslada ya sea a otra VM dentro del mismo servidor, o a otra VM en una máquina física distinta. Cuando una máquina física se queda sin recursos, también se realizan migraciones de VMs completas a otra máquina física que incluso podría estar en otro centro de datos (diferente región). Estas son tareas comunes con las que lidian todos los días los proveedores de recursos en la nube como Amazon, Apple, Google, Microsoft, entre otros.

Para el diseño de un modelo de un sistema o su arquitectura, se deben considerar varios factores como el costo, la complejidad, la velocidad, la portabilidad, la seguridad, entre otros. La arquitectura en la nube se diferencia de las arquitecturas locales en que 1) es altamente escalable, 2) es una entidad abstracta y aborda diferentes niveles de servicio, 3) aprovecha la economía de escala y 4) entrega los servicios de forma dinámica a través de la virtualización (Mishra *et al*., 2017). Cuando se arrenda un espacio virtual en la nube, no se reserva una máquina física completa, sino que se crea una máquina virtual dentro de una máquina física en un centro de datos, que bien podría tener otros cientos de máquinas virtuales.

Este artículo da una clara idea de la complejidad que tiene la tarea de implementar una arquitectura de alta disponibilidad localmente porque no solo

requiere de muchos conocimientos, sino también conlleva un alto costo. He aquí la importancia y los beneficios que se obtienen al utilizar los servicios de *cloud computing* como los que ofrece Amazon, porque AWS es quien se encarga de gestionar los recursos y escalarlos cuando es necesario. Además, el tiempo de implementación es sumamente rápido y a un menor costo comparado con las soluciones *on premise*.

Por otro lado, con este artículo se comprendió que el servicio de AWS Lambda puede garantizar el mismo nivel de disponibilidad que garantiza un *Load Balancer* y requiere de menos configuraciones. Si la arquitectura lógica de un sistema se adhiere a una arquitectura de microservicios, la tecnología *serverless* puede ser el complemento perfecto. Mientras el Balanceador de Carga distribuye las solicitudes entre varios nodos que deben ser configurados, una función Lambda puede crear varias instancias de sí misma sin tener que configurar más servidores.

Otra tecnología de particular relevancia cuando se habla de sistemas altamente disponibles es la segregación de funciones a través de una arquitectura orientada a microservicios y en pro de esta tecnología, existen los llamados contenedores que son pequeños servidores que se configuran específicamente para los requerimientos de un sistema, en este caso, de un microservicio. La tesis que se titula *Diseño e implementación de una Arquitectura escalable basada en Microservicios para un Sistema de gestión de Aprendizaje con características de red social*, desarrollado en 2017 por José Manuel de Paz Estrada y publicado por la Universidad San Carlos de Guatemala, propone una arquitectura de microservicios para desplegar una aplicación para la gestión de aprendizaje. La arquitectura planteada pretende garantizar una alta disponibilidad

y tolerancia a fallos de los servicios que ofrece la aplicación. Uno de sus componentes son los contenedores Docker, que son máquinas virtuales optimizadas para realizar una tarea específica.

Docker es un motor de contenerización que automatiza el empaquetado, envío y despliegue de aplicaciones de *software*. Son contenedores ligeros, portátiles y autosuficientes, que podrán ejecutarse en cualquier servidor (De Paz, 2017). Estos contenedores son totalmente compatibles con los servicios de *hosting* en la nube y, de hecho, muchos proveedores ponen a disposición de los desarrolladores imágenes de contenedores preconfiguradas con ciertos requerimientos y listas para levantarse.

En esta investigación el autor demostró el uso de estos contenedores y su integración con microservicios, así como la accesibilidad a través de una API Gateway. Esta arquitectura le permitió lograr una disponibilidad del 100 % de sus servicios y garantizó una alta tolerancia a fallos. Sin embargo, el prototipo del sistema que implementó es relativamente pequeño y se compone únicamente de dos módulos. El mismo autor concluye que para sistemas pequeños o de poco alcance, no se recomienda esta arquitectura porque la administración de la infraestructura se complica demasiado.

De esta tesis se concluye que no es viable utilizar contenedores Docker para el despliegue de las aplicaciones porque requieren demasiadas configuraciones y además el sistema que se quiere implementar se compone únicamente de dos módulos: ingreso de datos y presentación de resultados. Sin embargo, sí se adopta el diseño de las aplicaciones orientado a microservicios

de manera que cada módulo pueda operar de manera independiente y pueda incluso desplegarse en un contenedor.

Por otro lado, para el despliegue de la API que obtiene los resultados, se adoptó la tecnología *serverless* que es una alternativa a los contenedores y requiere de menos configuración y ninguna administración. Amazon ofrece el servicio de Lambdas que son procesadores de código en la nube y no requieren de ninguna administración del servidor. Simplemente se levanta el código de la API dentro de una Lambda y esta se encarga de ejecutarlo y devolver los resultados, garantizando siempre la escalabilidad y el pago únicamente por tiempo de procesamiento.

Para profundizar en las funciones Lambdas de AWS y los microservicios, se analizó el trabajo de Mikulas Zaymus, *Decomposition of monolithic web Application to microservices* que desarrolló en el 2017. El objetivo de este estudio es exponer los aspectos a tomar en cuenta para convertir una aplicación monolítica en una arquitectura basada en microservicios, así también como las herramientas necesarias para implementar dicha arquitectura. Resalta las ventajas y desventajas del uso de microservicios a comparación de una aplicación monolítica, luego describe una posible estrategia para la complicada tarea de descomponer una aplicación monolítica en microservicios, y por último explica el proceso de implementación y las herramientas que utilizó para garantizar alta disponibilidad en su aplicación prototipo que más adelante se va a detallar.

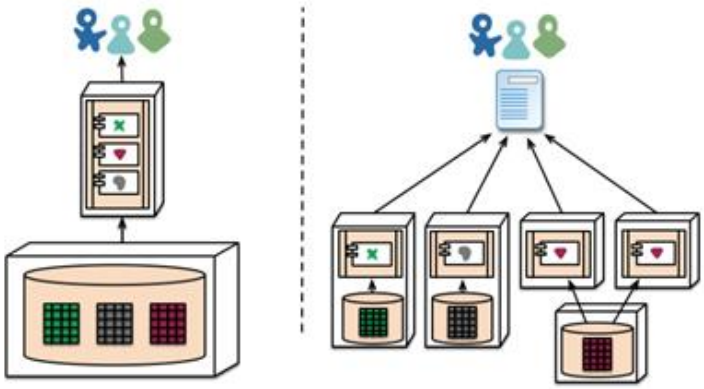
Zaymus optó por los servicios de Amazon para implementar la arquitectura ya que es actualmente la plataforma en la nube más madura y popular (Zaymus,

2017). Elegir la tecnología correcta para un proyecto nuevo es una de las tareas más difíciles ya que esta decisión puede facilitar el trabajo de los programadores como también puede traer dolores de cabeza en el futuro. Los servicios que usó Zaymus en su implementación fueron: Amazon S3, Amazon Route 53, Amazon DynamoDB, Amazon EC2, AWS Lambda, Amazon API Gateway y Amazon CloudFormation (Zaymus, 2017).

Además, diseñó una arquitectura orientada a microservicios para poder desplegarse fácilmente mediante funciones Lambdas de AWS. Una de las diferencias principales entre una aplicación monolítica y una compuesta por microservicios es que la monolítica corre en un proceso único mientras que cada microservicio corre en su propio proceso (Zaymus, 2017). Es altamente recomendado utilizar una base de datos independiente para cada microservicio, aunque existe la posibilidad de que varios microservicios puedan depender de la misma base de datos como se muestra en la Figura 4.

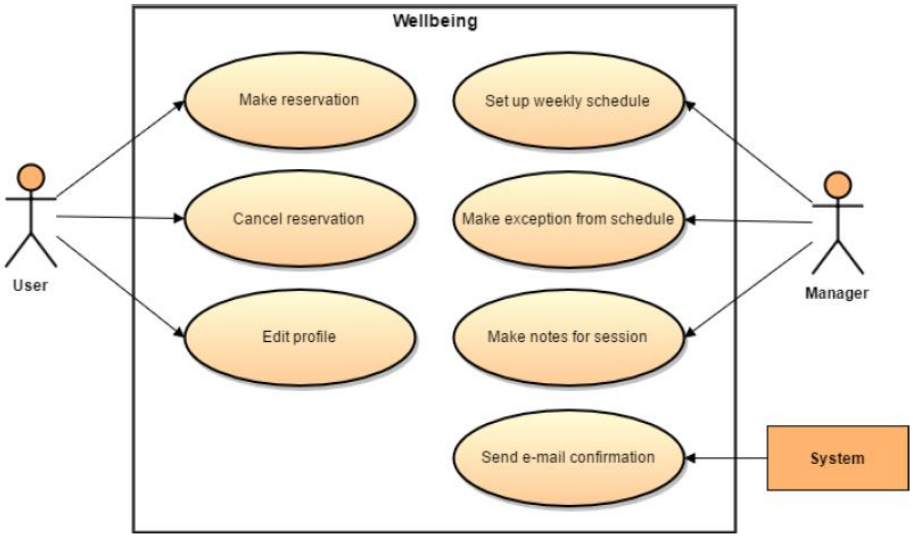
El proyecto de Zaymus se llama *Solteq Wellbeing* y consiste en una aplicación que facilita las citas y atención a los empleados de la empresa Solteq, que ofrece el servicio de masajes. La aplicación es usada por los trabajadores para realizar sus citas y por la masajista que llevan el control de estas. Los casos de uso de dicha aplicación se muestran en la Figura 5.

Figura 4. **BD aplicación monolítica vs. BD aplicación orientada a microservicios**



Fuente: Zaymus. (2017). *Decomposition of monolithic web Application to microservices*.

Figura 5. **Casos de uso del sistema Solteq Wellbeing**

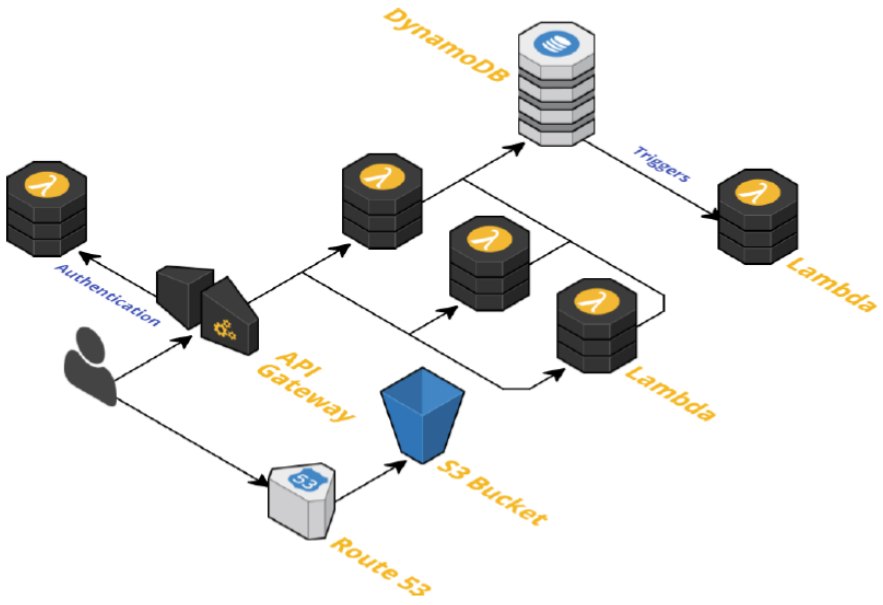


Fuente: Zaymus. (2017). *Decomposition of monolithic web Application to microservices.*

La aplicación *frontend* se desarrolló con Angular mientras que el *backend* fue escrito en NodeJS usando el *framework* Express. Para la base de datos se usó AWS DynamoDB, que es una base de datos orientada a documentos (Zaymus, 2017). Se utilizó Amazon S3 para alojar la página estática de la aplicación lo que provee una alta escalabilidad a un bajo precio. Amazon Route 53 se usó para gestionar las rutas entre cada servicio y para el registro de los dominios. Luego para desplegar los microservicios, Zaymus utilizó las funciones Lambda de AWS que permiten correr código sin necesidad de configurar ningún servidor, siendo totalmente escalable y toda la infraestructura y el ambiente es manejado por AWS (Amazon Web Services, Inc., 2019). Por último, se utilizó Amazon API Gateway para acceder a la lógica de negocio en las Lambdas y para autenticar a sus usuarios. Esto permite que cada microservicio pueda enfocarse en las necesidades de su dominio sin preocuparse de los derechos de acceso. Esto simplifica dramáticamente el código y reduce la complejidad en cada microservicio (Zaymus, 2017). La arquitectura completa de su implementación se muestra en la Figura 6.

Para el diseño de los microservicios, luego de determinar los dominios, dividió la aplicación monolítica en pequeñas partes que pueden operar de manera independiente. Estas partes podrían luego trabajarse de forma aislada y desplegarse en su propio microservicio. La primera solución en la nube podría ser correr una instancia de Amazon EC2 por cada microservicio y balancear el tráfico mediante un Load Balancer. Luego en caso de alta carga, puede agregarse una nueva instancia de Amazon EC2. Esta arquitectura no provee muchos beneficios en comparación a la solución monolítica existente y es muy complicado de gestionar desde el punto de vista del Load Balancer (Zaymus, 2017).

Figura 6. **Arquitectura AWS Lambda de Solteq Wellbeing**



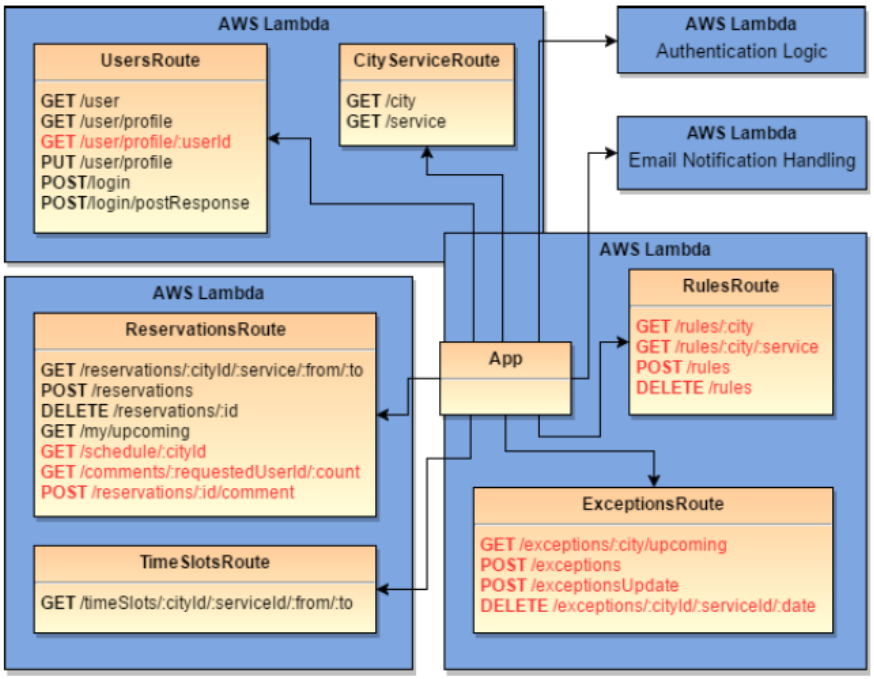
Fuente: Zaymus. (2017). *Decomposition of monolithic web Application to microservices.*

Una mejor solución es elegir el servicio en la nube AWS Lambda como ejecutor de los microservicios y así delegar a Amazon la gestión de los recursos y del balanceador de carga, siempre garantizando una alta disponibilidad y la posibilidad de escalar. Otro beneficio de AWS Lambda respecto a Amazon EC2 es que es más barato, debido a que en Amazon EC2 se paga por los recursos reservados mientras que en AWS Lambda se paga por el número de solicitudes y tiempo de ejecución de cada función (Zaymus, 2017).

Algunas partes lógicas de la aplicación que podrían estar en sus Lambdas respectivas, se decidió implementar en una sola función Lambda. La razón es que el código en una Lambda no reside en la memoria todo el tiempo, por lo que en llamadas prolongadas se carga desde el disco duro y eso penaliza el tiempo

de respuesta. En la Figura 7 se muestra el desglose final de la aplicación en microservicios alojadas en sus Lambdas respectivas.

Figura 7. **API de Solteq Wellbeing en AWS Lambda**



Fuente: Zaymus. (2017). *Decomposition of monolithic web Application to microservices.*

La investigación de Zaymus ayudó a comprender de manera más detallada la aplicación de los servicios que provee Amazon, pero el hallazgo más importante es el hecho de usar AWS Lambda en vez de Amazon EC2, tomando en cuenta las ventajas de una arquitectura sobre la otra y las necesidades del problema que se quiere resolver. En el caso de la API que recibe los votos, puede estar sin ningún problema en una instancia de EC2 ya que va a recibir muy pocas solicitudes. Por otro lado, en este proyecto se va a utilizar una base de datos

relacional, contrario a DynamoDB que fue utilizado por Zaymus. La razón es que se quiere aprovechar la ventaja que tiene un RDBMS para tener una redundancia de la base de datos mediante una configuración Master-Slave y además se quiere aprovechar las capacidades del RDBMS para garantizar transacciones ACID.

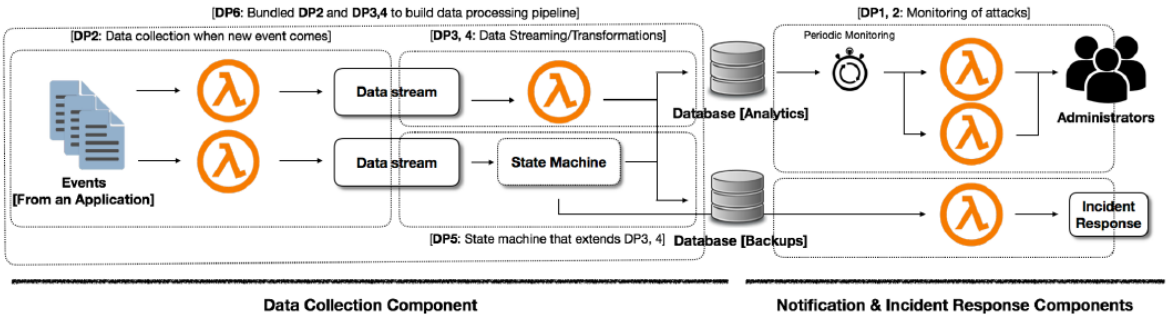
Por último, la investigación de Zaymus amplió los conocimientos acerca de la implementación de una arquitectura orientada a microservicios y su integración con los servicios de AWS. También se valora el conocimiento de que en una función Lambda se pueden desplegar varios microservicios y que las bases de datos pueden ser compartidas.

Otro aspecto importante de las FaaS o tecnología *serverless* es la seguridad y la confiabilidad de la ejecución del código. Al ser un servicio que funciona sobre internet, es susceptible a ataques informáticos como sucede con los servidores tradicionales. Posibles ataques podrían ser la alteración del código durante la subida (*man-in-the-middle*), infección del contenedor de la función con un *malware* para modificar su comportamiento, una denegación de servicio (DoS) o incluso una inyección de SQL. Los proveedores en la nube como Amazon, Microsoft o IBM manejan un modelo de responsabilidad-compartida respecto a la seguridad. En este modelo, el proveedor se encarga de la seguridad de la infraestructura mientras el consumidor es responsable de la seguridad de sus recursos y su información (Hong *et al*., 2018).

Sanghyun Hong, Abhinav Srivastava, William Shambrook y Tudor Dumitras publicaron en 2018 el artículo *Go Serverless: Securing Cloud via Serverless Design Patterns* y proponen una arquitectura compuesta por funciones Lambda para recopilar registros (*logs*), detectar actividades maliciosas y tomar acciones

preventivas. Identificaron 6 patrones de diseño para las arquitecturas Serverless: invocación periódica (intervalo definido), invocación basada en eventos (event-driven), transformación de datos (ETL o extract-transform-load), *streaming* de datos, Lambdas conectadas en una máquina de estados finitos y, por último, un patrón que combina los anteriores llamado Bundled Pattern (Hong *et al*., 2018). La arquitectura que proponen para el tratamiento de eventos sospechosos se muestra en la Figura 8 y es una combinación de los patrones antes mencionados.

Figura 8. **Arquitectura Lambda para la detección de actividades maliciosas**



Fuente: Hong *et al*. (2018). *Go Serverless: Securing Cloud via Serverless Design Patterns.*

Varios de los componentes de AWS generan eventos para varios propósitos como el análisis de rendimiento, rastreo de errores o análisis de seguridad. La data generada por los eventos puede pasar por Lambdas para agregar información extra como datos de geolocalización, luego otra Lambda hace *streaming* de los eventos a una BD. Cuando el *streaming* de la data falla, una máquina de estados captura el error e invoca otra Lambda para volver a intentar. Para analizar la data de los eventos, una Lambda monitorea periódicamente la

BD para identificar posibles actividades sospechosas y alertar a los administradores como, por ejemplo, IPs fuera de rango, varios intentos fallidos de inicio de sesión, entre otros. Por último, una Lambda pendiente de los incidentes, pueden accionar inmediatamente ante cualquier evento sospechoso bloqueando todo el tráfico tanto de entrada como de salida de la Lambda comprometida (Hong *et al*., 2018).

Adicionalmente, los autores hacen un análisis del costo y la escalabilidad de dicha arquitectura, considerando que también podría implementarse con instancias de EC2. Llegaron a la conclusión de que para tareas que requieren una corta latencia (5-10ms), es difícil correrlas en una función Lambda ya que estas tienen una alta latencia de arranque (50-100ms) por lo que puede impactar en el tiempo de respuesta (Hong *et al*., 2018). Por otro lado, el tiempo que dura una ejecución y la periodicidad de las llamadas también incide en el análisis del costo ya que, si una tarea dura poco y es impredecible, una arquitectura *serverless* es más barata porque se paga únicamente por tiempo de uso. Sin embargo, si las tareas duran más de un minuto y son muy frecuentes, el costo de las Lambdas puede subir drásticamente por lo que convendrá mejor tener instancias fijas de EC2 y pagar un monto fijo mensual (Hong *et al*., 2018).

Otra de las limitaciones de las funciones Lambda es que poseen un tiempo límite de ejecución y una vez se alcanza dicho tiempo, AWS detiene la ejecución y automáticamente se pierde la información y el estado. Considerando esta limitación, no es recomendable el uso de esta tecnología para tareas que requieren mucho tiempo de procesamiento y muchos recursos en CPU y memoria, ya que AWS dispone únicamente de 512MB de memoria por cada llamada (Hong *et al*., 2018).

En conclusión, este artículo resalta la importancia de la seguridad del código que se va a ejecutar en las funciones Lambda. Este proyecto no contempla este amplio campo de investigación ya que las funciones Lambda a utilizar se encargan únicamente de devolver los resultados de las elecciones y las BDs que usarán serán únicamente de lectura, por lo que se evita una posible alteración de la información. Sin embargo, si es necesario tomar medidas de seguridad en la instancia de EC2 dónde estará alojada la BD y el microservicio para ingresar los resultados para evitar cualquier intrusión.

Respecto al análisis del costo y las limitaciones de las funciones Lambda que plantearon Hong *et al*., afortunadamente la tarea de las funciones Lambdas de este proyecto consiste básicamente en realizar algunas consultas a la BD y luego devolver los resultados, lo cual no debería ser tardado, por lo que una arquitectura *serverless* es más conveniente que una arquitectura de instancias EC2.

Al inicio de esta sección se mencionó que son varios los componentes que pueden afectar o mejorar el rendimiento de un sistema que exige una alta disponibilidad. Una de ellas es la capa de datos y para profundizar en la arquitectura que se va a implementar en este proyecto, se analizó el artículo *Building Scalable and Resilience Database System to Mitigate Disaster and Performance Risks*, desarrollado en el año 2018 por Aryan Wibowo, Diana, Mohammad Subekti y Hendro. Este *paper* aborda el tema de construir bases de datos resilientes, tolerables a fallos y que garanticen una alta disponibilidad. Normalmente el plan de Recuperación ante Desastres (*Disaster Recovery*) y la Optimización del desempeño, se manejan por separado. Para la tolerancia a

fallos, algunos optan por realizar copias de seguridad en el mismo servidor, sin embargo, este procedimiento impacta en el desempeño de la base de datos (Wibowo *et al*., 2018).

Las tecnologías de bases de datos evolucionan rápidamente a través del tiempo. Bases de datos de código abierto ampliamente utilizados como MariaDB y MySQL incluyen ya soluciones para garantizar alta disponibilidad, que van desde simples copias de seguridad (*backup*) hasta la replicación de datos o almacenamiento compartido. Los procesos complejos de *backups* tradicionales están quedando atrás y ahora varios métodos de replicación son disponibles como una estrategia de *backup* y como una opción de *Disaster Recovery* (Wibowo *et al*., 2018).

Aunque este artículo se enfoca más en la tolerancia a fallos, el modelo planteado también provee grandes beneficios a la alta disponibilidad. La pérdida de la integridad significa que la data proveída no es confiable y la pérdida de disponibilidad simplemente significa que la base de datos es inaccesible. Estas dos situaciones pueden afectar las operaciones del negocio por lo que es de vital importancia ser consideradas dentro del Plan de Recuperación ante Desastres (Wibowo *et al*., 2018).

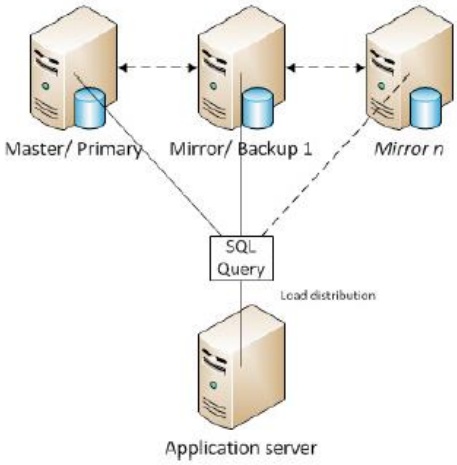
El enfoque comúnmente utilizado para soportar la tolerancia a fallos es tener dos sitios idénticos: uno primario y otro secundario que deben estar en zonas diferentes. Desafortunadamente, tener dos sitios agrega un costo significativo a IT y tener ese gasto mensual por desastres que raramente van a ocurrir, no parece justificable. Es por esta razón que se buscan alternativas para garantizar una alta disponibilidad y tolerancia a fallos, y los sistemas de bases de datos

modernas (RDBMS) vienen dotadas con funciones de replicación que permiten propagar las transacciones a otras copias de la BD.

La replicación de bases de datos se trata de mecanismos para guardar la información en múltiples locaciones. Usualmente, el término *Master/Slave* se usa para ilustrar que cualquier cambio o actualización en la BD *Master* (primaria), es replicada en la BD *Slave* (secundaria). La replicación se da en una sola dirección lo que significa que un cambio en la BD secundaria no será reflejado en la BD primaria, en otras palabras, la BD secundaria es solo de lectura. La ventaja de esta arquitectura es que mejora la disponibilidad de los datos sin tener mucho impacto sobre los costos. Los *backups* ya no afectarán el desempeño porque ahora se hacen sobre la BD secundaria (Wibowo *et al*., 2018).

En la Figura 9 se muestra este modelo de replicación y se puede observar que las consultas se realizan tanto en *Master* como en *Slave*, con la diferencia de que en *Slave* solo se aceptan consultas de tipo *SELECT*. El proceso de replicación es extremadamente rápido y consume recursos insignificantes. Los RDBMs integran funciones para implementar esta arquitectura y es posible replicar localmente o replicar en otro servidor o incluso en otra región.

Figura 9. **Modelo de replicación Master-Slave**



Fuente: Wibowo *et al*. (2018). *Building Scalable and Resilience Database System to Mitigate Disaster and Performance Risks.*

Evidentemente, el modelo de replicación propuesto por este artículo se adapta perfectamente a las necesidades que presenta el Sistema de tabulación y la Página de resultados de las elecciones de Guatemala. Los datos pueden ingresarse en una BD *Master*, luego se replican en una BD *Slave* con una consistencia eventual y la API que obtiene los resultados los lee directamente de esta BD. La aplicación para mostrar los resultados no necesita realizar ninguna escritura o actualización, lo cual es precisamente el objetivo de una BD configurada como *Slave* que es solo de lectura.

2. JUSTIFICACIÓN

Este proyecto contribuye al área de investigación *Sistemas para impulsar el uso de Cloud Computing en los negocios*, ya que propone una arquitectura en la nube aplicable a sistemas que demandan una alta disponibilidad y continuidad de negocio, aprovechando los servicios que ofrece Amazon Web Services (AWS) y explotando las últimas tecnologías de la industria IaaS (*Infraestructure as a Service*). Además, fomenta el uso de microservicios que también contribuyen a la disponibilidad de los sistemas de información y encajan precisamente con la reciente tecnología *serverless*, o mejor conocido como Lambda en el mundo de AWS.

Hoy en día sistemas *Legacy* (escritos en lenguajes antiguos o lenguajes deprecados) y *server-rendering* (el servidor devuelve tanto la data como la vista) siguen soportando la información de las empresas. Sin embargo, a través del tiempo la demanda de información crece exponencialmente y los sistemas informáticos deben evolucionar si quieren seguir satisfaciendo la necesidad de los clientes. En este proyecto se fomenta la implementación de aplicaciones como *Single Page Application* (SPA) y la manipulación de la información a través de REST APIs que se encargan únicamente de recibir y devolver la data, mejorando así el tiempo de respuesta.

Cuando se trabaja con sistemas donde la disponibilidad en todo momento es un factor crítico para las operaciones del negocio, uno de los factores más importantes a considerar es la arquitectura a nivel de infraestructura que se va a

implementar para prestar los servicios. Aquí es donde el modelo de servicios en la nube IaaS cobra importancia porque implementar localmente todas las tecnologías necesarias para garantizar escalabilidad y alta disponibilidad, implica costos exorbitantes y requiere de expertos que posean amplios conocimientos en las 4 capas inferiores del modelo OSI (transporte, red, enlace y físico). Además, su implementación toma mucho tiempo y lo más probable es que no se alcance el nivel de calidad y fiabilidad que ofrecen los proveedores de servicios en la nube como Amazon. Estos proveedores resuelven todos estos inconvenientes y permiten a las empresas concentrarse más en su negocio que en su infraestructura. Los servicios de AWS ofrecen muchos beneficios como la escalabilidad, levantar servidores en menos de un minuto, seguridad, confiabilidad y pago únicamente por uso (Amazon Web Services, Inc., 2019).

La relevancia de esta investigación está en que se hará uso de las tecnologías más recientes de *cloud computing* y se hará una demostración de cómo se pueden integrar dichas tecnologías para ofrecer una solución escalable, fiable y de bajo costo. Esta misma arquitectura puede aplicarse en otros contextos que exigen también una alta disponibilidad, como la banca electrónica, los sistemas *e-commerce*, sistemas hospitalarios, sistemas contables, entre otros.

El principal beneficiado que se espera de esta investigación es el departamento de informática del Tribunal Supremo Electoral de Guatemala, ya que, aunque los 2 sistemas que se van a desarrollar en este proyecto no vayan a adoptarse e implementarse para las elecciones reales de Guatemala, la arquitectura que se presenta puede servir a sus ingenieros para que lo puedan

implementar ellos mismos en un futuro o les sirva de base para comenzar, ya que es totalmente escalable.

Por otro lado, este estudio provee un vasto conocimiento sobre los principales servicios que ofrece Amazon y cómo estos se pueden integrar entre sí. Una de las razones por la que no se adopta la tecnología en la nube es la falta de conocimiento e inseguridad sobre los resultados que se pueden obtener de estos servicios. Además, Amazon ofrece una amplia gama de servicios que en la mayoría de los casos termina confundiendo a los encargados de IT y no son capaces de identificar cuáles servicios son los que realmente necesitan.

Por último, esta investigación también puede servir como base para realizar nuevos estudios de *cloud computing* o agregar mejoras a la arquitectura presentada, por ejemplo, la tolerancia a fallos no está considerada. Esto es otro campo de investigación que aporta también a la alta disponibilidad de los sistemas de información. Implementar un sistema tolerante a fallos, implica tener redundancia de todos los posibles puntos de fallo: servidores de aplicación, servidores de servicios REST API, servidores de base de datos, proveedores de *cloud*, proveedores de red de fibra óptica, proveedores de energía eléctrica, entre otros.

3. ALCANCES

3.1. Perspectiva investigativa

El presente trabajo de investigación posee en general un enfoque explicativo y descriptivo. El alcance investigativo desglosado en los objetivos específicos es como se describe a continuación:

- Configuración de la base de datos con redundancia Master-Slave: durante la revisión documental, posee un enfoque explicativo porque se da a conocer en qué consiste esta arquitectura y cuál es su contribución en una arquitectura de alta disponibilidad. Luego, durante la presentación de los resultados tiene un alcance con enfoque descriptivo, ya que se proveerá una documentación de los pasos a seguir para implementar esta arquitectura.
- Modularización de las aplicaciones: este objetivo es el resultado de la arquitectura de Microservicios. En la revisión documental se explican todos los conceptos relacionados con este patrón de diseño y en la presentación de los resultados se describen los componentes lógicos de la arquitectura y su comunicación con las bases de datos.
- Arquitectura en AWS: de igual manera, durante la fase investigativa posee un alcance explicativo ya que se documenta cada uno de los servicios que se utilizarán para configurar la arquitectura en AWS y se explica el papel que desempeña cada uno dentro de la arquitectura de alta disponibilidad. Por otro lado, en la presentación de resultados tiene un alcance con enfoque

descriptivo porque se detalla el proceso de configuración y se describe el rendimiento de la arquitectura en cada iteración de pruebas.

3.2. Perspectiva técnica

A continuación, se desarrolla el alcance del proyecto desde la perspectiva técnica, tratando de cubrir los objetivos específicos planteados.

- Se usó una base de datos relacional y el RDBMS seleccionado es MySQL. En este RDBMS se trabajó la configuración para tener una BD maestra (primaria) y dos esclavas (secundarias), donde las BDs secundarias son réplicas de la BD primaria.
- Las aplicaciones web (ingreso de resultados y presentación de resultados) se implementaron como *Single Page Application (SPA)* con el *framework* Angular basado en TypeScript y se usó Bootstrap como *framework* de CSS. Cada aplicación se trabajó de manera independiente. La aplicación para la tabulación de los votos contempla también la gestión de usuarios y la autenticación. Esto significa que maneja un token de sesión (JWT).
- La comunicación entre la aplicación web del sistema de tabulación y la API correspondiente se realiza sobre el protocolo SSL/TLS para garantizar la confidencialidad y autenticidad de la información. En otras palabras, la API posee un certificado de seguridad que permite encriptar la comunicación con HTTPS.
- Para la implementación de la arquitectura física se usaron los servicios de EC2 y AWS Lambda de Amazon. Cada BD (primaria y secundarias) se alojó en una instancia de EC2; el microservicio que permite el ingreso de los resultados se configuró también en una instancia de EC2; en las funciones

Lambda se desplegaron los microservicios para la consulta de los resultados y consulta de la información pública; las aplicaciones web se desplegaron en AWS CloudFront usando el servicio de AWS Amplify.

- Las APIs se desarrollaron en el lenguaje NodeJS y se usó Express como *framework*. Las APIs para consultar los resultados y consultar la información pública se implementaron como funciones que pueden ejecutarse en AWS Lambda. Por otro lado, la API que permite el ingreso de los resultados también se diseñó como un microservicio, pero como una API RESTful levantada con un servidor Nginx en una instancia de EC2.
- Las pruebas de carga se realizaron mediante la librería de NodeJS llamada Artillery (Artillery.io), la cual permitió ejecutar las pruebas de manera sincronizada en contenedores de AWS Fargate.

3.3. Perspectiva de los resultados esperados

Los entregables y resultados del proyecto son los siguientes:

- Un *cluster* de tres bases de datos en MySQL en una configuración de replicación *Master-Slave*. Las BDs secundarias deben tener una consistencia eventual con la BD primaria y debe permitir únicamente operaciones de lectura (*readonly*).
- Se presentan dos aplicaciones web como SPAs, una para la tabulación de los resultados y otra para la presentación (página de resultados). La aplicación del sistema de tabulación permite también la gestión de usuarios y requiere que las solicitudes estén autenticadas.

- Una arquitectura 100 % funcional en AWS capaz de atender hasta 2000 consultas de resultados a cada 10 segundos, garantizando así una alta disponibilidad.
- Documentación del proceso de implementación, desde el diseño de los microservicios (arquitectura lógica) hasta el proceso de configuración de los servicios en Amazon (arquitectura física).
- Archivos de configuración de Artillery de las pruebas realizadas.

4. MARCO TEÓRICO

4.1. Cloud Computing

Gracias a la expansión del internet y la rápida evolución de las tecnologías de procesamiento y almacenamiento, los recursos de computación se han vuelto más accesibles y poderosos. Gracias a estos avances tecnológicos ha emergido el modelo de *cloud computing* que consiste en proveer recursos de *hardware* y/o *software* sobre internet de manera automatizada aprovechado la tecnología de la virtualización. Este modelo es atractivo porque permite a los negocios escalar sus recursos conforme aumenta su demanda (Avram, 2014).

Los servicios que se basan en el modelo de computación en la nube pueden clasificarse en tres grupos principales: Software como Servicio (SaaS), Plataforma como Servicio (PaaS) e Infraestructura como Servicio (IaaS). Cada uno abstrae una capa del soporte tecnológico que una empresa necesita en sus operaciones para entregar una solución a través de internet. La aparición de este modelo revoluciona totalmente la forma en que los servicios de Tecnología de la Información se inventan, implementan, escalan, mantienen y pagan (Avram, 2014).

Existen dos modelos principales de despliegue en la nube: público y privado, aunque ultimadamente las empresas optan por una combinación de ambas creando una nube híbrida.

4.1.1. Nube pública

Una nube pública es un conjunto de componentes de *software* e infraestructura física provista y administrada por una empresa que se dedica específicamente a mantener grandes centros de datos en puntos estratégicos alrededor de todo el mundo para ofrecer servicios de computación en la nube a los negocios y las instituciones. Todos los componentes en *hardware* y *software* que soportan la infraestructura son propiedad del proveedor y es compartida por los consumidores gracias a la virtualización. Las ventajas más significativas que ofrece una nube pública sobre una nube privada son: costos bajos por el aprovechamiento de la economía de escala y el modelo de pago por uso, no requiere mantenimiento a nivel de infraestructura, escalabilidad casi ilimitada y mayor tolerancia a fallos gracias a la disponibilidad de varios centros de datos en regiones distintas.

4.1.2. Nube privada

Por su parte, una nube privada es aquella que se usa única y exclusivamente para una organización, siendo la misma quién la implementa y la administra. Las nubes privadas suelen implementarse por entidades que buscan tener un control total sobre su infraestructura, tales como instituciones financieras, entidades del gobierno y cualquier organización mediana o grande que busque una máxima protección de su información (Microsoft, 2019). Los beneficios principales que se obtienen con una nube privada son: provee mayor control sobre los recursos, se puede implementar mayores niveles de seguridad, también se puede escalar con facilidad si la organización tiene la capacidad económica. Sin embargo, hay que tener presente de que el hecho de tener una

nube privada no es garantía de seguridad total de la información ya que eso va a depender de la capacidad y experiencia del implementador.

4.1.3. Nube híbrida

Una nube híbrida combina infraestructura local con nubes públicas para optimizar las tareas de procesamiento, sin descuidar la protección y privacidad de la información sensible. Un ejemplo de caso de uso es usar la nube pública para procesar grandes volúmenes de datos que no requieren mayor protección y usar la nube privada para manejar la información sensible de la empresa. En este esquema pueden estar involucrados varios proveedores dependiendo del servicio que se quiere subcontratar y lo que se busca es mayor flexibilidad y rentabilidad.

4.1.4. Proveedor de servicios en la Nube

Un proveedor de servicios en la Nube o CSP por sus siglas en inglés (*Cloud Service provider*), es una compañía que ofrece recursos y servicios computacionales en masa sobre internet para facilitar a las empresas implementar sus sistemas de información. Estos proveedores ofrecen una gran variedad de servicios, pero se categorizan principalmente en IaaS, PaaS y SaaS. En general, los CSP poseen *data centers* distribuidos en todo el mundo para garantizar mayor disponibilidad, tienen procesos automatizados para que los consumidores puedan adquirir sus servicios en línea, aprovechan la economía de escala para reducir sus precios y cobrar únicamente por tiempo de uso o por transacción, poseen procesos automáticos de replicación y tolerancia a fallos,

disponen de APIs para interactuar con sus servicios desde aplicaciones *backend* y comúnmente sus servicios son multiplataforma y multilenguaje.

Existen muchas compañías que ofrecen estos servicios en todo el mundo, pero son contados los que dominan el mercado. Entre estos sobresalen: Amazon con AWS, Microsoft con Azure y Google con Google Cloud Plataform. Otros no tan dominantes pero que tienen su parte en el mercado son: IBM Cloud, Oracle Cloud, Rackspace, Salesforce y Apple iCloud. Mientras el negocio principal es la provisión de IaaS, en los últimos años se han explorado nuevas áreas como Machine Learning, Inteligencia Artificial (AI), Contenedores de aplicaciones, funciones serverless (FaaS), Internet de las Cosas (IoT) y business intelligence (BI).

Aunque el negocio de *cloud computing* sigue creciendo y tiene mucha madurez, todavía existen desafíos que no se cubren totalmente. La seguridad de la información en la nube no está garantizada al 100 % y, de hecho, los CSP siguen el modelo de responsabilidad compartida dónde ellos aseguran los recursos mientras el consumidor debe responsabilizarse de la seguridad de los datos. La tolerancia a fallos tampoco se garantiza en su totalidad y los proveedores disponen de un service-level-agreement (SLA) donde especifican el porcentaje de nivel de servicio, por ejemplo 99.9 % *uptime*. Otro dilema que los proveedores deben resolver es la compatibilidad *multi-cloud* ya que muchas empresas implementan nubes híbridas y pueden depender de varios proveedores, por lo que necesitan integrar sus servicios y realizar migraciones entre nubes.

4.1.5. Infraestructura como servicio

Conocido comúnmente como IaaS por sus siglas en inglés, es un modelo de computación en la nube que provee infraestructura y almacenamiento bajo demanda sobre internet. Los recursos son virtualizados y esto incluye servidores, redes, DNS, balanceadores de carga y espacio de almacenamiento. En vez de comprar, instalar, integrar y luego administrar su *hardware*, las empresas rentan estos recursos mientras lo van necesitando (Hurwitz *et al*., 2012). La base de la computación en la nube es la virtualización ya que permite separar los recursos y los servicios de la capa física de manera que se pueden crear muchas máquinas virtuales en una misma máquina física (Hurwitz *et al*., 2012).

Los servicios de tipo IaaS se dividen en 3 categorías: 1) públicas, que es infraestructura compartida y cualquiera con una tarjeta de crédito puede rentarla, 2) privadas, que es propio de una organización y se maneja en una red privada, 3) híbridas, que es una combinación de infraestructura privada con infraestructura pública.

Las características principales de IaaS que valen la pena resaltar son: se usa en modo de alquiler mediante una subscripción ya sea mensual, trimestral, semestral o anual; los usuarios lo obtienen por sus propios medios a través de un portal y no necesitan contactar con el departamento de IT del proveedor; escalan de manera dinámica ya que los usuarios pueden aumentar los recursos cuando lo necesitan, lo que implica que los proveedores poseen procesos de migración automatizados; garantizan un porcentaje de nivel de disponibilidad a través de un contrato (SLA); el cargo se calcula con base al uso (número de instancias, cantidad de almacenamiento, número de transacciones procesadas,

entre otros) y a servicios adicionales como seguridad, soporte o monitoreo avanzado (Hurwitz *et al*., 2012).

4.1.6. Plataforma como servicio

También conocido como PaaS (*platform as a service*), este modelo de servicio se encuentra una capa más arriba de IaaS y ofrece un ambiente para los desarrolladores para crear, correr y administrar aplicaciones y servicios. El beneficio principal de PaaS es que ahorra a los desarrolladores el trabajo de configurar un ambiente de desarrollo. Entre los servicios de este tipo podemos encontrar sistemas operativos, bases de datos, lenguajes de programación, ambientes de pruebas (*testing*), flujos de entrega continua (*deployment*), ambientes colaborativos, entre otros. Algunas empresas estandarizan sus herramientas y procesos de desarrollo de aplicaciones mediante un PaaS, siguiendo buenas prácticas para lograr una mejor eficiencia, mejor productividad y reducir los costos (Hurwitz *et al*., 2012).

El uso de PaaS mejora el ciclo de vida del proceso de desarrollo ya que normaliza las herramientas y las versiones utilizadas por todos los desarrolladores, evitando así inconvenientes a la hora de integrar, probar y construir una aplicación. Esta abstracción elimina el proceso de configuración de ambiente o actualización de componentes, estandariza las versiones de las herramientas de desarrollo, reduce los errores de compatibilidad y asegura consistencia en el ciclo de vida de desarrollo (Hurwitz *et al*., 2012).

4.1.7. Software como servicio

Es una aplicación creada y alojada por un proveedor, estando disponible para cualquier persona que tenga acceso a internet. Un SaaS se implementa ya sea sobre un PaaS o directamente sobre un IaaS. Estas aplicaciones son bastante comunes y se usan frecuentemente como Gmail, Facebook, Twitter, Dropbox, Instagram, entre otros. Aunque estas son aplicaciones simples, también se pueden encontrar aplicaciones complejas como de contabilidad, de facturación, de inventario, para monitoreo, CRMs, manejo de nóminas, de *marketing*, entre otros. Cualquier aplicación accedida desde internet es un SaaS.

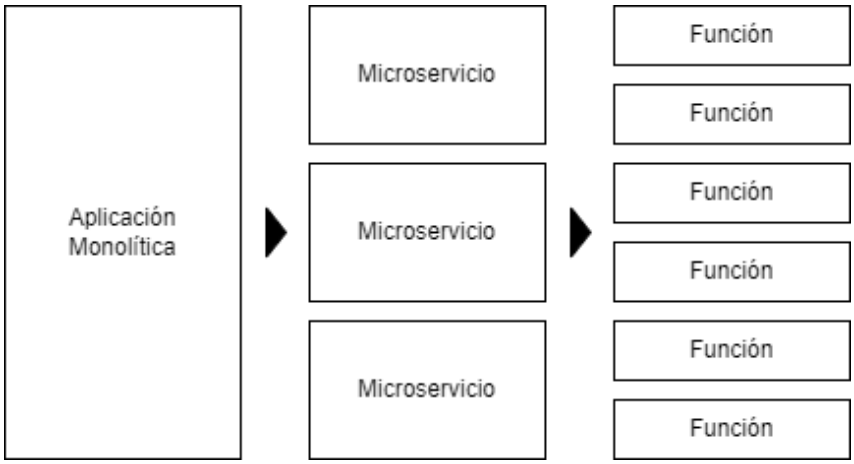
Con el *software* tradicional, se debe comprar, instalar, mantener y comprar las licencias correspondientes. Cuando hay una actualización, hay que repetir todo el proceso. Sin embargo, con un SaaS se obtiene únicamente un usuario y una subscripción si es necesaria, y ya se puede acceder a la aplicación desde cualquier dispositivo. Las actualizaciones las realiza el proveedor.

4.1.8. Función como servicio

También se le identifica como FaaS (*function as a service*) o *serverless*, es relativamente un nuevo concepto que fue introducido por primera vez en 2014 por AWS y consiste en una tecnología que permite implementar Arquitecturas *serverless*. Este servicio permite la ejecución de código en respuesta a eventos sin necesidad de implementar o mantener una infraestructura compleja. Con esta tecnología, el desarrollador se olvida de la configuración de un servidor, contenedor o ambiente de ejecución, simplemente carga a la nube su código escrito en forma de función y configura un evento que ejecutará dicha función.

Con esta arquitectura, nunca habrá un tiempo en que el servidor (el *hardware* reservado por la función porque no existe un servidor como tal) se queda sin trabajo porque una vez termina la función, automáticamente se liberan los recursos. Esto reduce los costos ya que el modelo de pago es únicamente por tiempo de uso y/o por transacción.

Figura 10. **Aplicación monolítica vs. microservicios vs. FaaS**



Fuente: Cloudflare. (2019). *What is Function-as-a-Service (FaaS)?*

La clave para construir esta arquitectura es modular un sistema o una funcionalidad en funciones que se pueden ejecutar de manera independiente. Este modelo se usa principalmente para construir microservicios. La Figura 10 muestra una comparativa entre este modelo, una arquitectura de microservicios y una aplicación monolítica. Como puede notarse, la arquitectura *serverless* ofrece mayor escalabilidad. El proveedor es quien administra el equipo que ejecuta cada función y es quien garantiza la disponibilidad (Cloudflare, Inc., 2019). Si en determinado momento se requiere atender 100 solicitudes, el

proveedor instancia 100 copias del servicio y si la demanda se reduce a 2 solicitudes, destruye las que no se necesitan (Baez, 2020).

Las funciones pueden llamarse en cadena, por ejemplo, una función podría solo agregar una transacción a una cola y otra función se encarga de ejecutarla. La arquitectura *serverless* ahorra el trabajo de configurar servidores, pero presenta la desventaja de que es difícil monitorear el proceso de ejecución porque no se tiene control sobre el *hardware* donde se ejecuta (no se tiene acceso a una consola). Por otro lado, dividir una aplicación en pequeñas funciones independientes es una tarea compleja y si no se hace bien esta abstracción, se puede terminar con una función con cientos de líneas de código, haciendo más difícil el proceso de *debugging* y la lectura del código, o incluso afectando el rendimiento de la aplicación.

4.2. Amazon web services

Abreviado y mejor conocido como AWS, es un conjunto de servicios de computación e infraestructura en la nube que provee la empresa tecnológica y estadounidense Amazon, Inc. Provee recursos computacionales y servicios (*cloud service provider* o CSP) para desplegar aplicaciones en la nube en cuestión de minutos, siguiendo el modelo de pago por uso y con posibilidad de escalar mientras la demanda aumenta. AWS evita tener que levantar una infraestructura física local que puede llevar semanas o incluso meses y requiere de alta inversión. En cambio, ofrece soluciones que funcionan sobre una red de escala global y gracias a la economía de escala, los costos se reducen significativamente. De hecho, AWS puede reducir los costos hasta un 80 % en

comparación con la implementación local equivalente (Amazon Web Services, Inc., 2014).

En AWS se puede correr cualquier componente que se pueda correr en *hardware* físico: páginas web, aplicaciones, bases de datos, aplicaciones móviles, campañas por correo, análisis de datos distribuidos, almacenamiento de multimedia y redes privadas (Amazon Web Services, Inc., 2014). Para interactuar con estos servicios, AWS provee muchas herramientas para los desarrolladores: 1) AWS Management Console, consiste en una consola gráfica que se puede acceder sobre internet, 2) AWS Command Line Interface (CLI), es una herramienta que dispone de varios comandos que pueden ejecutarse en una consola para manipular múltiples servicios de AWS, 3) Software Development Kits (SDKs), son librerías de clases y herramientas que se pueden agregar dentro de una aplicación para manejar los recursos de AWS programáticamente, disponibles para los lenguajes de programación más populares y para el desarrollo móvil en IOS y Android, y 4) Query APIs, son *endpoints* de bajo nivel que permiten interactuar con ciertos servicios de AWS mediante solicitudes sobre HTTP (Amazon Web Services, Inc., 2014).

En la actualidad, AWS ofrece una plataforma altamente confiable, escalable y de bajo costo que impulsa a cientos de miles de empresas en 190 países de todo el mundo. Con centros de datos (*data centers*) en EE.UU., Europa, Brasil, Singapur, Japón y Australia, los clientes de todo tipo de industria aprovechan los beneficios de AWS: bajo costo gracias a la economía de escala y el modelo de pago por uso, elasticidad ya que pueden incrementar o reducir sus recursos según su demanda, flexibilidad ya que es el cliente quién define la arquitectura lógica que tendrá su aplicación así como las herramientas donde se va a

desplegar, y por último es seguro ya que AWS cuenta altas certificaciones y posee múltiples capas de seguridad que garantizan la integridad de los datos (Amazon Web Services, Inc., 2019).

4.2.1. Amazon EC2

EC2 es el acrónimo de Elastic Compute Cloud y es uno de los primeros servicios que Amazon lanzó al mercado, el cual vino a revolucionar la forma en que las empresas implementan su infraestructura tecnológica, ofreciendo una alta disponibilidad, escalabilidad y seguridad. En cuestión de minutos se puede configurar un servidor en EC2 y todo a través de la web en forma de autoservicio como suele llamarse (Golden, 2013). El término autoservicio hace referencia a que no existe ninguna negociación o comunicación con un colaborador de Amazon sino todo se realiza a través de su plataforma web mediante procesos automatizados. Años atrás, para levantar un servidor y poder ser accedido sobre internet, había que tener dinero suficiente para comprar uno, realizar la compra, esperar a la entrega, instalarlo, obtener una IP pública y por último conectarlo a la red. Este proceso podía tomar entre 3 y 6 meses (Golden, 2013).

EC2 se basa en la virtualización y cada instancia de EC2 es una máquina virtual (VM) que actúa como computadora real con un sistema operativo (OS) y todo el *software* necesario para ejecutar alguna tarea. En la terminología de EC2, una máquina virtual que está corriendo es una instancia y cuando no está corriendo, se dice que es una imagen. La diferencia entre la virtualización de EC2 y la que realizan productos como VMware o VirtualBox, es que EC2 solo ocupa el espacio que necesita el OS y el *software* instalado mientras que en una VM se ocupa todo el espacio que se le configuró. Por ejemplo, en una VM clásica si se

le configura 10GB de disco duro y el OS junto con el *software* solo requieren de 1GB, se está dejando sin uso 9GB y el OS anfitrión no puede utilizar ese espacio (Golden, 2013).

Esto es crucialmente importante porque implica una reducción de costos ya que en EC2 no se tiene que pagar por espacio sin uso o cuando la instancia no está corriendo. Por otro lado, los costos también van a depender del tipo de instancia que se tiene. Las instancias varían por la cantidad de tres tipos de recursos: poder de procesamiento, tamaño de memoria y capacidad de almacenamiento (Golden, 2013).

4.2.2. Amazon Lambda – Serverless

En la última década, la computación en la nube ha cambiado radicalmente la forma en que se implementan servicios de IT y ha permitido que varias empresas emergentes implementen aplicaciones disruptivas y escalables como Spotify, Dropbox o Netflix. El valor agregado de *cloud computing* es que permite a los desarrolladores concentrarse únicamente en su código y en implementar las características del sistema, evitando las tareas que no le agregan valor a su trabajo como administrar o escalar una infraestructura, instalar parches en los sistemas operativos o mantener actualizado sus herramientas de desarrollo (Poccia, 2017).

Los servicios en la nube incluyen servidores virtuales, espacio de almacenamiento, redes, balanceadores de carga, entre otros. Aunque estas tecnologías en la nube han reducido considerablemente el trabajo del desarrollador, aún requieren de alguna configuración previa para ejecutar o correr

una aplicación. Incluso, usando un contenedor para levantar un ambiente con herramientas como Docker, aún requiere de inversión de tiempo en definir el archivo de configuración, administrar las versiones, construir y desplegar una imagen. Un PaaS puede facilitar este trabajo si se implementan procesos automatizados de *deployment* (DevOps), sin embargo, también requiere de una implementación y una fase de pruebas. La tarea se vuelve aún más compleja si se le agrega el problema de disponibilidad, escalabilidad y seguridad (Poccia, 2017).

AWS Lambda hace una abstracción de estas tareas y permite a los desarrolladores subir su código agrupado en funciones que luego podrán ejecutarse de manera independiente en la plataforma de AWS. De esta manera, el programador ya no tiene que levantar un servidor, instalar un *framework* o un conjunto de librerías, ni tiene que preocuparse por la disponibilidad y la escalabilidad. Cada función tiene su propia configuración que permite usar estándares de seguridad de AWS para definir qué puede hacer y sobre qué recursos (Poccia, 2017). Gracias a esta tecnología, es posible implementar una Arquitectura Serverless dirigida por eventos (*event-driven architecture*). El término Serverless hace referencia al hecho de que el programador se olvida de la configuración y administración de un servidor.

Las funciones Lambda pueden ser invocadas directamente desde una aplicación o pueden subscribirse a eventos generados por otros recursos y cada vez que ocurra el evento, automáticamente se ejecuta la función. Por ejemplo, cuando un archivo es subido a S3 o un nuevo registro es ingresado a la BD, una función Lambda puede reaccionar a estos eventos y registrar una entrada en una bitácora. Si una imagen fue subida, una función podría automáticamente crear

los *thumbnails* (imagen de pequeña resolución para previsualización); si un registro fue modificado en la BD, una función podría replicar los cambios en un *data warehouse* (Poccia, 2017). Siguiendo este patrón de diseño se pueden implementar aplicaciones altamente escalables dirigidas por eventos y a bajo costo ya que se paga únicamente por tiempo de consumo.

AWS Lambda ejecuta las funciones usando de manera eficiente los recursos, lo cual permite un interesante e innovador modelo de costo: a) Se paga por el número de invocaciones, y b) Se paga por el tiempo de ejecución de todas las invocaciones dependiendo de la cantidad de memoria asignada a cada función (Poccia, 2017). Las simplificaciones introducidas por las funciones Lambda, construyendo las partes de una aplicación bloque por bloque, permiten ser más eficientes en dónde invertir el tiempo y más rápidos en llevar nuevas características a los usuarios (Poccia, 2017).

4.3. Microservicios

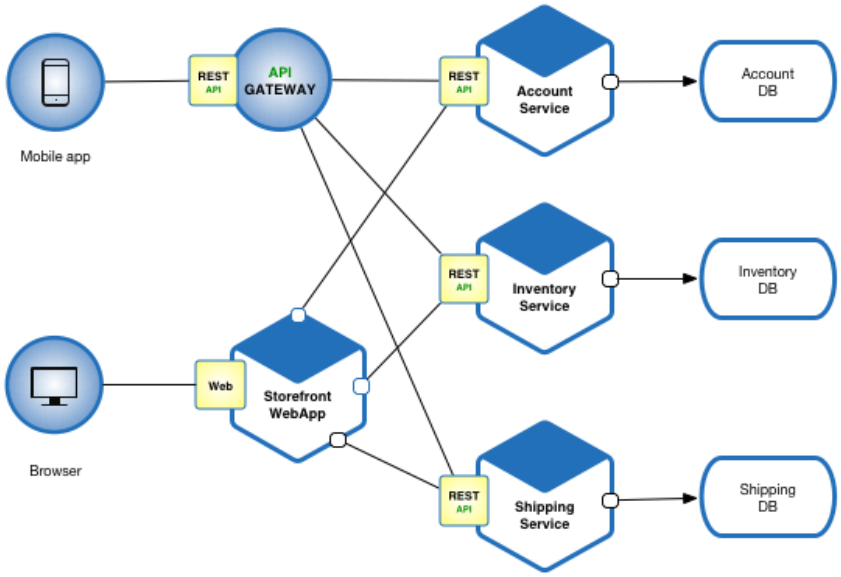
A través del tiempo, se han descubierto nuevas formas y nuevos patrones de diseño de cómo resolver los problemas del mundo real con sistemas web sofisticados y orientados al usuario, pero también buscando facilitar el trabajo de los programadores, desde el diseño de la arquitectura lógica, hasta la entrega del sistema funcional al usuario final. Luego, también se ha buscado facilitar cada vez más la entrega continua de actualizaciones, el mantenimiento del código y la escalabilidad de la arquitectura. Además, gracias a la proliferación de los servicios computacionales en la nube y la virtualización, se han encontrado nuevas formas de construir arquitecturas escalables y resilientes.

Los microservicios juegan un papel importante en esta innovación ya que consisten en servicios granulares y autónomos que se pueden desarrollar y desplegar independientemente. Si se diseñan de la manera correcta, se puede construir un sistema flexible, escalable y mantenible. El código base de un sistema puede crecer de manera inmensurable que eventualmente va a ser muy difícil de mantener y de resolver los errores. Los microservicios siguen el principio de responsabilidad única donde cada servicio está limitado a una función específica y se enfoca en hacer una sola cosa, pero lo hace bien (Newman, 2015).

4.3.1. Arquitectura de microservicios

En un principio, los sistemas de información basados en web utilizaban una arquitectura monolítica donde todas las funciones y los servicios estaban encapsulados en una sola aplicación. Esto limita la escalabilidad y flexibilidad del sistema porque lo obliga a escalar de forma vertical y eventualmente alcanzar un límite. Escalar verticalmente consiste en aumentar los recursos de procesamiento, almacenamiento y memoria RAM. En un sistema monolítico, un pequeño cambio afecta a todo el sistema e implica una actualización completa, un error en un componente puede bloquear el acceso a las demás funciones, hay una gran cantidad de código y cada vez será más difícil de mantener, el proceso de *deployment* es tardado y por último, dificulta el trabajo colaborativo.

Figura 11. **Arquitectura de microservicios**



Fuente: Richardson. (2019). *What are microservices?*

Una arquitectura de microservicios resuelve todos estos problemas modulando lo más granular posible un sistema en pequeños servicios autónomos modelados por un dominio del negocio. Cada microservicio posee las siguientes características: posee bajo acoplamiento, se puede mantener/probar/desplegar por separado, implementa una sola capacidad de negocio (una sola función), es implementado y mantenido por un pequeño equipo, e incluso puede contar con su propia BD tal como se muestra en la Figura 11. Esta arquitectura permite la entrega rápida, frecuente y confiable de aplicaciones grandes y complejas (Richardson, 2019).

El diseño de una arquitectura orientada a microservicios requiere identificar los dominios principales del negocio y dividirlas en partes pequeñas, ya que la

característica principal de un microservicio es que es totalmente independiente del resto, tanto en operación como en implementación. De hecho, cada microservicio debería tener su propia base de datos. La comunicación entre microservicios se da mediante HTTP, REST o algún tipo de RPC (Zaymus, 2017).

En la literatura informal, suele confundirse microservicios con SOA (*Service Oriented Architecture*) y aunque ambos siguen el mismo principio, hay una gran diferencia en el nivel de abstracción de cada uno. En SOA de hecho, no existe el término microservicio sino simplemente posee servicios, los cuales son muy generales y pueden encapsular un módulo completo. Por su parte, los servicios en una arquitectura de microservicios son más granulares y totalmente autónomos, es decir, cada uno funciona como una caja negra y su implementación es totalmente independiente del resto. SOA usa un Enterprise Service Bus (ESB) para manejar la comunicación entre los servicios y los clientes mientras los microservicios se comunican directamente entre ellos o directamente con el cliente. El ESB puede ser un único punto de fallo dejando inhabitado todos los servicios de SOA. Los microservicios buscan el desacoplamiento mientras SOA busca la reusabilidad.

Entre las ventajas principales que se obtienen de una arquitectura orientada a microservicios se pueden destacar:

- Desacoplamiento**:** cada microservicio es totalmente independiente del resto por lo que se puede mantener y escalar de manera individual (Kappagantula, 2019). También facilita el mantenimiento del código y la integración de un nuevo miembro al equipo.

- Autonomía**:** cada microservicio posee su propio equipo de desarrollo autónomo (Kappagantula, 2019). El tamaño del equipo puede variar, pero una regla bastante adoptada es que el número de desarrolladores no debe exceder el número de personas que pueden alimentarse con dos pizzas (Zaymus, 2017).
- Gobernanza descentralizada: cada equipo tiene la libertad de usar el lenguaje y las herramientas que más le convenga o que más se adecuen a la tarea (Kappagantula, 2019). No existen restricciones de tecnologías porque los microservicios se comunican mediante APIs.
- Agilidad y entrega continua: el desarrollo aislado de los microservicios facilita bastante las tareas de DevOps (Newman, 2015).
- Resistencia a fallos: debido a que cada microservicio corre en su propio proceso o contenedor, si uno falla los otros seguirán funcionando (Newman, 2015).

La implementación de microservicios presenta muchos desafíos entre los cuáles se pueden destacar dos: el diseño de la arquitectura en términos del tamaño que debe tener un microservicio y la orquestación e integración de estos. No existe un método o una métrica para saber qué tan granular debe ser un microservicio, sino se debe enfocar en el modelo lógico del dominio de la aplicación y asegurarse que cada microservicio posea sus propios datos. La cohesión y alta dependencia entre dos componentes indica que deben ir dentro de un solo microservicio.

Por otro lado, una arquitectura de microservicios presenta ciertos inconvenientes que deben tomarse en cuenta:

- Al ser un sistema distribuido, puede dificultar la comunicación entre microservicios, sobre todo cuando una operación requiere de una cadena de llamadas síncronas entre varios microservicios (Huston, 2019).
- Al incrementar la cantidad de microservicios, la arquitectura crece en complejidad y los desarrolladores deben lidiar con la tolerancia a fallos, la latencia de red y el balanceo de carga (Huston, 2019).
- Si los microservicios son muy granulares, se puede caer en el error de tener nano servicios, tener lógica y esfuerzo duplicados. Esto también puede aumentar la cantidad de llamadas entre microservicios y a su vez impactar en el tiempo de respuesta (Huston, 2019).

Una arquitectura compuesta por microservicios puede incluir una API Gateway que es una interfaz que proporciona un punto único de entrada para todos los microservicios, es decir, este se encarga de recibir todas las solicitudes y redirigirlas al microservicio que corresponde. Vale la pena mencionar que este componente en la arquitectura puede representar un mayor tiempo de respuesta y/o convertirse en un cuello de botella.

4.3.2. REST API

REST es el acrónimo de *Representational State Transfer* y es un estilo de comunicación web Cliente-Servidor que define un conjunto de reglas para implementar servicios web. Un servicio web que se implementa siguiendo esta arquitectura se le llama RESTful API. El nombre hace referencia a la forma en que el servidor intercambia información con el cliente y viceversa, mediante una representación del recurso en un formato específico, el cuál puede ser JSON, XML o HTML. Un recurso es la representación de un objeto o un componente del

negocio, por ejemplo, un “Cliente”. Cada recurso se accede mediante un URI (*Uniform Resource Identifier*) único. Comúnmente una API REST implementa *endpoints* para manipular un recurso como Crear, Leer, Actualizar y Eliminar (CRUD). Los verbos HTTP correspondientes son POST, GET, PUT y DELETE. Algunos desarrolladores utilizan el verbo PATCH para actualizar un registro en vez de PUT y la diferencia está en cómo el servidor procesa la data que recibe. Con un PUT reemplaza por completo el recurso almacenado por la nueva versión recibida, mientras que con un PATCH aplica únicamente los cambios enviados por el cliente.

REST es un estilo de arquitectura y no está amarrado específicamente al protocolo HTTP, sin embargo, las características propias de HTTP se ajustan perfectamente a la especificación de REST. Además, se aprovecha el control de seguridad que ya posee HTTP para autenticar la comunicación, desde una autenticación con usuario y contraseña, hasta certificados de seguridad (Newman, 2015).

Una API RESTful se adhiere a las siguientes condiciones:

- Arquitectura cliente-servidor: el servidor y el cliente debe estar totalmente desacoplados. Al separar la interfaz de usuario (UI) de la manipulación de la información, se mejora la portabilidad del UI y se mejora la escalabilidad reduciendo los componentes del servidor (Fielding, 2000).
- Operaciones stateless**:** cada solicitud debe tener toda la información necesaria para ser interpretada por el servidor y no necesita de ningún contexto almacenado previamente. En otras palabras, cada solicitud se

procesa de manera independiente. Si se necesita manejar un estado, lo debe realizar el cliente (Fielding, 2000).

- Cacheo de recursos (caching)**:** el servidor puede guardar en caché el estado de los recursos para optimizar el tiempo de respuesta, a menos que se le indique explícitamente que no lo haga (Fielding, 2000).
- Interfaz uniforme (UI)**:** la arquitectura define las siguientes cuatro restricciones: identificación de los recursos mediante URIs, manipulación de los recursos a través de representaciones, mensajes auto descriptivos y uso de hipermedia para la ingeniería del estado de la aplicación (Fielding, 2000). Este último requerimiento hace referencia a la navegación mediante enlaces que usan los navegadores y sugiere que el servidor debe incluir dentro del estado de un recurso los enlaces que conllevan a otras acciones.
- Sistema basado en capas**:** el cliente no debe saber si se está comunicando con el servidor final o con un intermediario. De esta manera, podría haber un proxy o un balanceador de cargas en medio y la comunicación no debe verse afectada (Fielding, 2000).
- Código bajo demanda (opcional)**:** normalmente el servidor retorna un XML o un JSON, sin embargo, también puede devolver código ejecutable (Fielding, 2000).

4.4. Disponibilidad de las bases de datos

Tener una arquitectura de microservicios orquestadas dinámica y eficientemente no es garantía de un sistema de alta disponibilidad. La forma en que se almacena y se obtiene la información también afecta considerablemente el tiempo de respuesta de un sistema. Es importante distinguir entre disponibilidad y durabilidad de la información ya que dependiendo de las

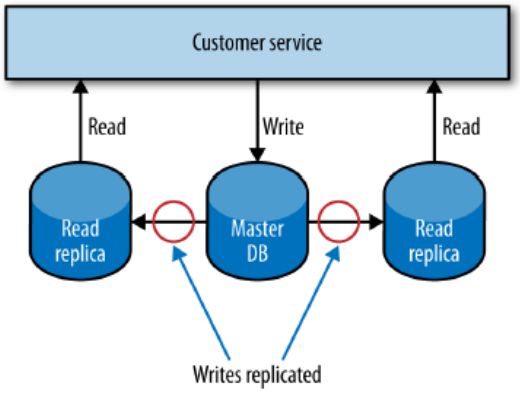
necesidades que se quieren fortalecer, se puede dar prioridad a una característica u otra, y a su vez puede restringir la arquitectura para persistir los datos.

Como ejemplo, se puede tomar la decisión de tener un proceso de *backup* que constantemente está guardando una copia de la información y al momento que la BD tenga fallos, la información no se pierde, pero el sistema quedará inactivo durante cierto tiempo. Un modelo común es tener una réplica de la BD principal y cuando esta presenta fallos, se hace el cambio a la BD secundaria, y aunque se garantiza la disponibilidad de la información y por consiguiente la continuidad de los servicios, la base de datos primaria pierde todas las transacciones que entran mientras está abajo (Newman, 2015).

4.4.1. Escalar la lectura con una arquitectura Master-Slave

En un Sistema de Manejo de Base de Datos Relacional (RDBMS) como MySQL o Postgres, la información puede ser copiada de un nodo primario a una o más réplicas. Es usado comúnmente para mantener una copia segura de la data, pero también puede utilizarse para escalar la lectura. Como se observa en la Figura 12, un servicio puede escribir directamente a un nodo, pero distribuir las lecturas a través de las réplicas. La réplica ocurre justo después de la escritura en la BD primaria, por lo que habrá un instante en que los nodos secundarios son inconsistentes hasta que se completa la replicación. A esta configuración se le conoce como consistencia eventual y en sistemas dónde no tiene efectos negativos sobre los servicios consumidores, es una estrategia común para escalar (Newman, 2015).

Figura 12. **Usando réplicas para escalar la lectura**



Fuente: Newman. (2015). *Building Microservices.*

4.4.2. Escalar la escritura con una arquitectura sharding

Escalar la capacidad de lectura en las bases de datos relacionales es trivial, sin embargo, escalar la escritura es mucho más complicado. Un enfoque consiste en utilizar una arquitectura *sharding* que es una técnica de "particionado horizontal" dónde se tienen múltiples tablas llamadas particiones con el mismo esquema y mediante una función *hash* se decide a qué partición se debe guardar una línea. Las particiones son autónomas y no comparten recursos, incluso pueden estar en localizaciones diferentes (Newman, 2015). Estas y otras capacidades son ofrecidas por sistemas como Casandra, Mongo o Riak.

La complejidad de la arquitectura *sharding* está en las consultas. Obtener un solo registro es sencillo porque mediante alguna función *hash* se puede saber en qué *shard* se encuentra y luego se obtiene del *shard* correspondiente. Sin embargo, cuando las condiciones de la consulta se extienden a través de varios *shards*, se necesitará ya sea realizar la consulta por cada *shard* y luego unificar

los resultados, o tener una réplica de lectura dónde estén todos los registros (Newman, 2015).

5. PRESENTACIÓN DE RESULTADOS

Como se menciona en el título de este trabajo, se busca diseñar e implementar un prototipo de arquitectura en AWS que soporte una alta carga de usuarios concurrentes en la página de consulta de resultados del TSE, esto sin afectar la operatividad del sistema de tabulación de votos. Bajo este contexto, antes de provisionar algún servicio en AWS o escribir una línea de código de las aplicaciones web de prototipo, se procedió primero con el análisis y diseño de la arquitectura lógica del sistema completo.

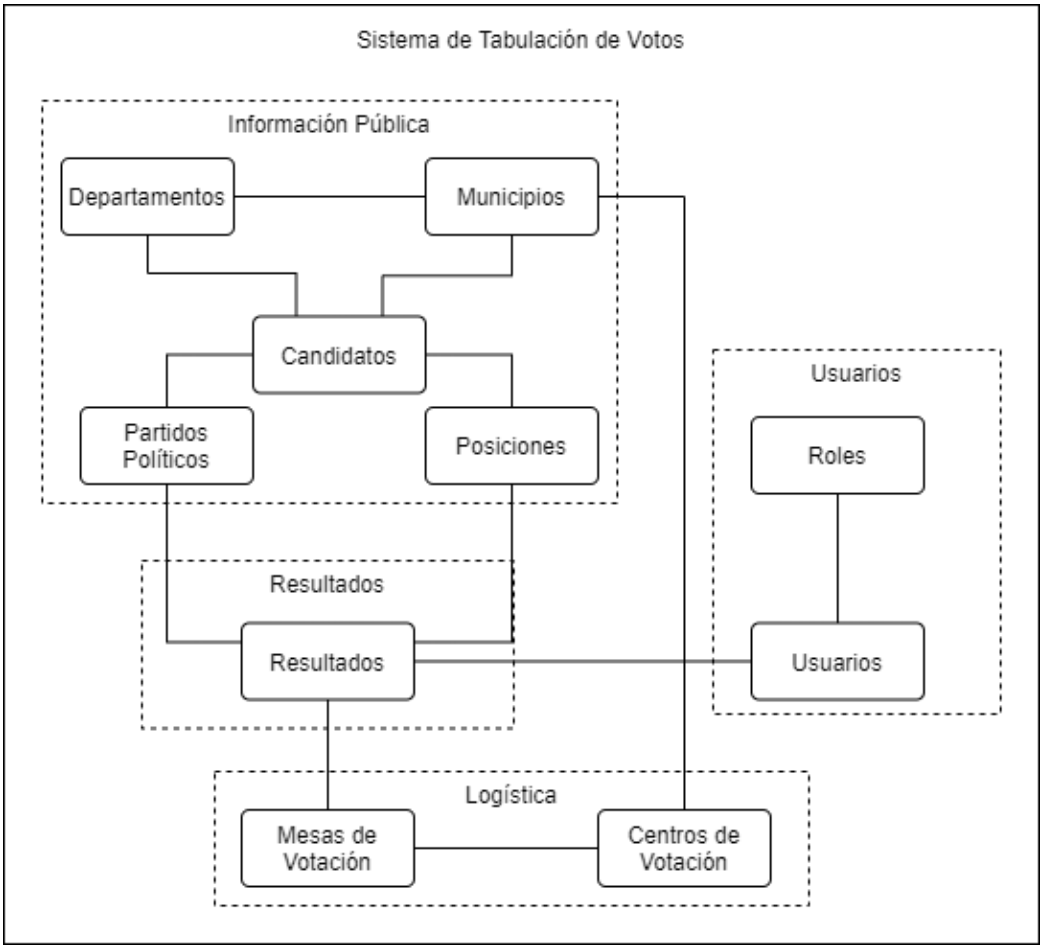
Recalcando el alcance del proyecto, se concentra únicamente en la digitalización de los resultados de la votación y en la presentación de estos a los usuarios de internet desde una página web. No contempla la digitalización de las actas de las mesas de votación. Además, asume que el proceso de tabulación se realiza desde un solo lugar (ciudad de Guatemala) y desde pocas terminales (3 computadoras máximo), es decir, no existe el riesgo de sobre carga. La consideración de la alta concurrencia es únicamente en la página de resultados.

5.1. Análisis y diseño de microservicios

Con la premisa de implementar una solución siguiendo un patrón de microservicios, se procedió con el análisis de los procesos de negocio, los componentes y actores involucrados en el sistema completo. Para facilitar esta tarea, se utilizó el método de *Domain Driven Design* (DDD por sus siglas en

inglés) para identificar los dominios del sistema completo, las entidades que la componen y los tipos de usuarios que interactúan con el mismo (Evans, 2003).

Figura 13. **Diagrama de modelo de dominio**

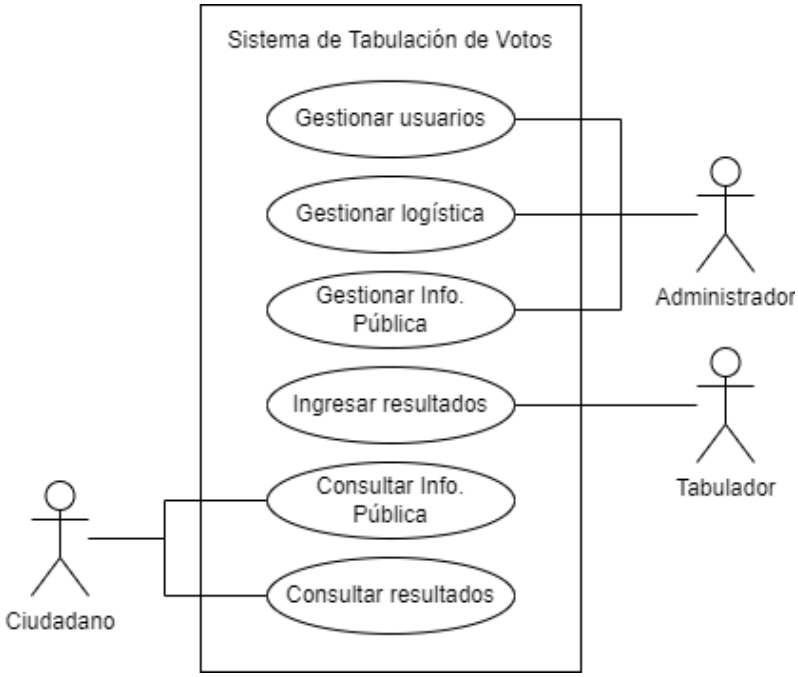


Fuente: elaboración propia, utilizando diagrams.net.

La Figura 13 muestra las diferentes entidades identificadas, sus relaciones y lo más importante, los contextos aislados que agrupan las entidades que comparten un mismo lenguaje y dominio. Aterrizado a la arquitectura de

microservicios, este diagrama da un acercamiento sobre los posibles módulos que pueden definirse como un microservicio. Las entidades de cada dominio poseen una alta cohesión mientras cada dominio tiene bajo acoplamiento con los otros dominios.

Figura 14. **Diagrama de casos de uso**

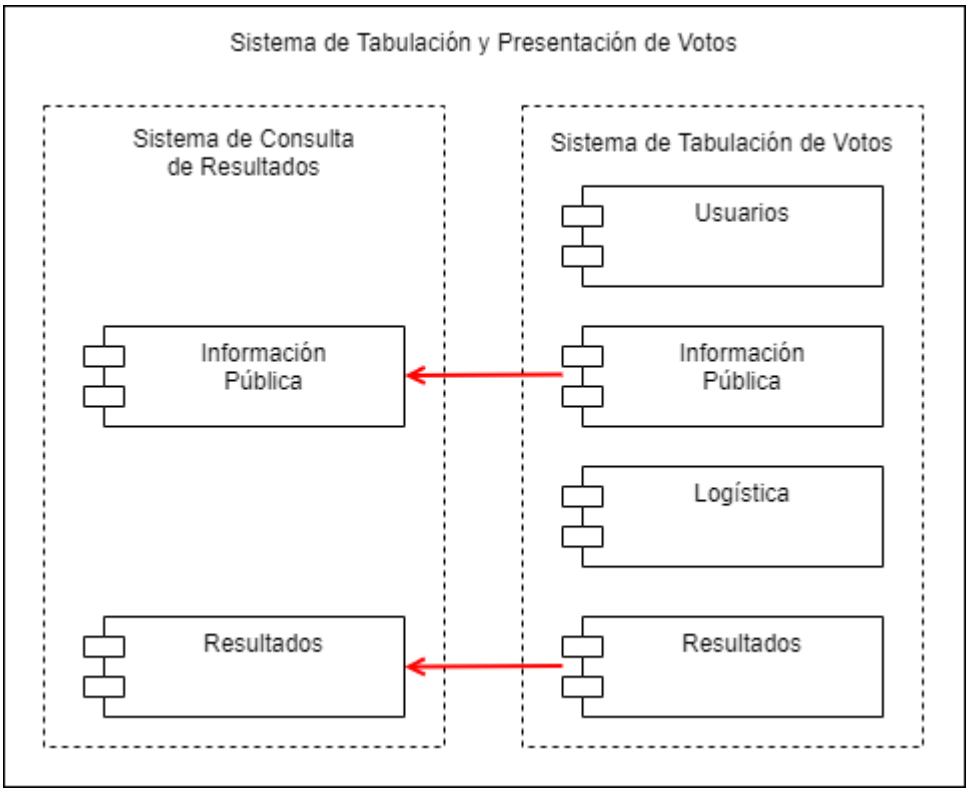


Fuente: elaboración propia, utilizando diagrams.net.

Para lograr una mejor comprensión de los procesos de negocio que debe soportar el sistema, se trabajó también el diagrama de casos de uso que se observa en la Figura 14. Tomando como base el diagrama de modelo de dominio, se mapearon los contextos como módulos independientes del sistema con los cuales interactúan los usuarios. El usuario “Administrador” es el encargado de gestionar a los usuarios del sistema, ingresar la información pública y crear los

centros y mesas de votación. Por su parte, el usuario “Tabulador” es quien se encarga de ingresar los registros de votos al sistema. Por último, el usuario que representa a la población en general es quién está interesado en conocer los resultados preliminares del proceso electoral e implícitamente necesitará obtener la información pública para filtrar y agrupar los resultados.

Figura 15. **Diagrama de módulos por casos de uso**

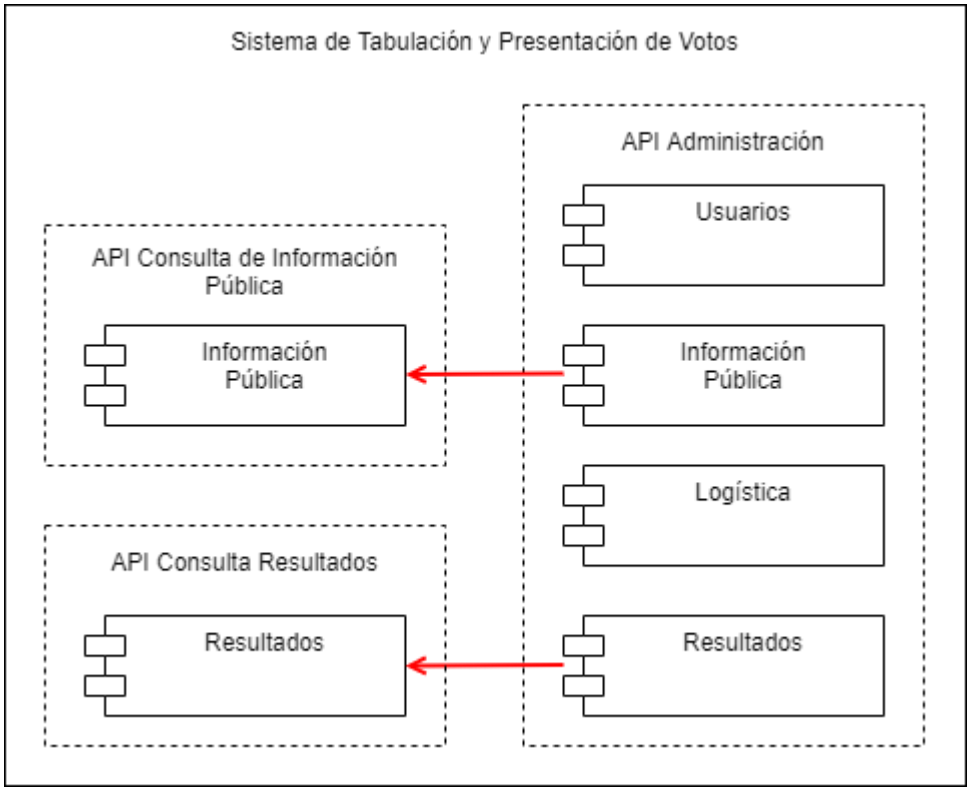


Fuente: elaboración propia, utilizando diagrams.net.

La Figura 15 muestra el diagrama de módulos con base en los casos de uso identificados anteriormente y como puede observarse, el sistema completo se puede dividir en dos módulos independientes: el sistema de consulta de

resultados y el sistema de tabulación. El primero únicamente necesita consultar la información, mientras el segundo es el que permite su ingreso.

Figura 16. **Diagrama de microservicios**



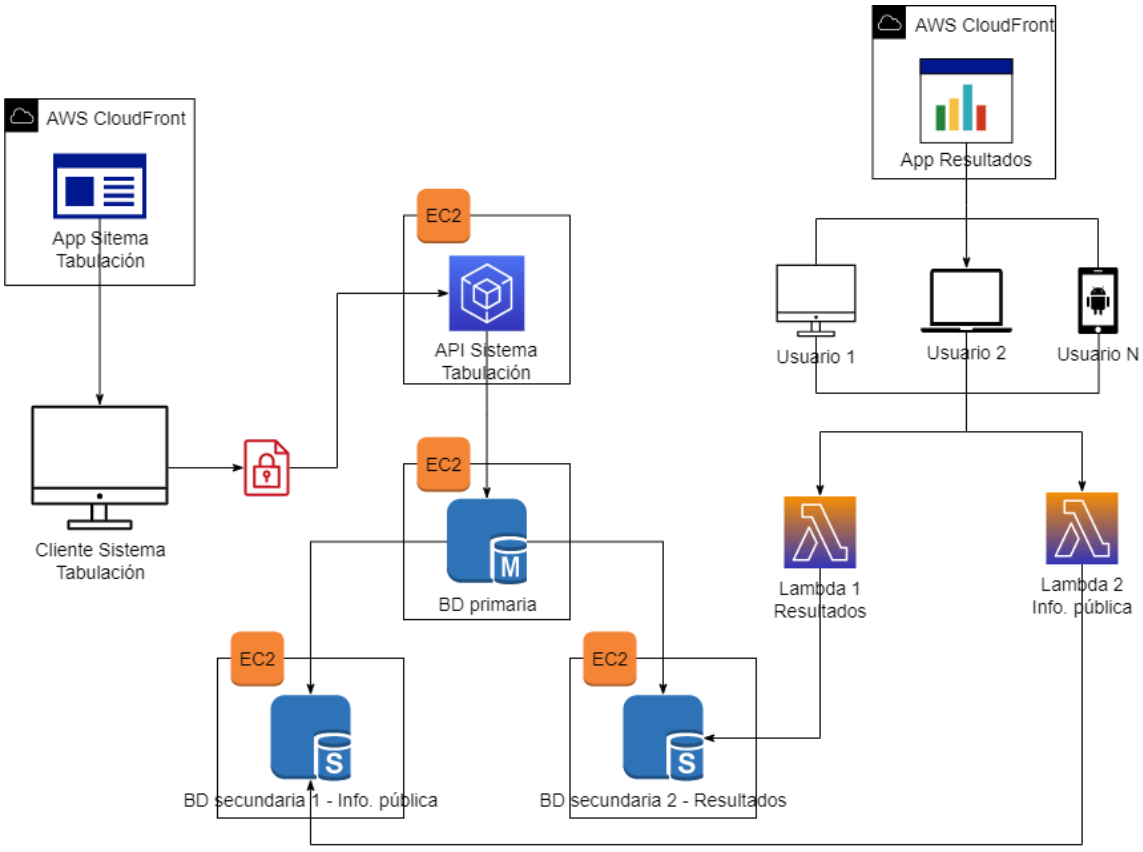
Fuente: elaboración propia, utilizando diagrams.net.

Finalmente, el diagrama de la Figura 16 muestra los microservicios que se lograron identificar de acuerdo con el dominio de las entidades, los casos de uso y los módulos desacoplados. Los microservicios se implementan como APIs totalmente independientes que permiten a las aplicaciones consultar y manipular la información de las bases de datos.

5.2. Diseño de la arquitectura de alta disponibilidad en AWS

Una vez se identificaron los procesos de negocio y estos fueron mapeados a un módulo de la arquitectura lógica, se procedió con el diseño y la diagramación de la arquitectura física que va a soportar cada proceso y va a permitir administrar la alta concurrencia de usuarios que van a consultar los resultados del proceso electoral. La Figura 17 muestra el diagrama de la arquitectura completa y los diferentes servicios de AWS que se utilizaron.

Figura 17. **Arquitectura de alta disponibilidad en AWS**



Fuente: elaboración propia, utilizando diagrams.net.

Del diagrama de la Figura 15 se logra identificar dos aplicaciones que pueden operar de forma independiente y aislada, ajustándose perfectamente a una arquitectura de microservicios. Para aumentar la disponibilidad de la información, se configuró un *cluster* de tres bases de datos, una BD maestra que de ahora en adelante se va a llamar la BD “primaria” y las otras dos como réplicas de la BD maestra, a las que se van a nombrar de ahora en adelante como bases

de datos secundarias o de réplica. En inglés se le conoce como arquitectura de replicación Master-Slave.

Tanto el sistema web para la tabulación de los votos como la página web para consultar los resultados consolidados, se implementaron como SPAs (*Single Page Application*) y se desplegaron (*deployment*) usando el servicio AWS Amplify que automatiza la distribución de las aplicaciones a través de la red de AWS CloudFront.

La aplicación para la tabulación de resultados obliga a los usuarios a autenticarse para acceder y manipular los datos. Esta aplicación permite alimentar todas las entidades (tablas) del sistema, tarea que debe realizar el Administrador antes del día de las elecciones. El usuario Tabulador ingresa un registro de votos totalizados y la información se encripta mediante el protocolo SSL/TLS antes de transmitirse sobre la red. En otras palabras, el servidor que expone la API para la tabulación de resultados cuenta con un certificado de seguridad para garantizar la integridad y confidencialidad de la información. La API se hospeda en una instancia de EC2 con recursos mínimos porque no necesita atender muchas solicitudes. La API al recibir la petición para guardar un registro de votos, verifica el *token* de sesión y si es válida, procede a guardar el registro en la BD primaria.

Todas las transacciones que se ejecutan sobre la BD primaria, se replican en las dos BDs secundarias manteniendo una consistencia eventual, es decir, durante un lapso mínimo de tiempo habrá una diferencia entre las bases de datos. Las BDs secundarias poseen la misma configuración y se alojan en instancias de

EC2 con las mismas características, aunque cuentan con más recursos de computación para soportar la alta concurrencia de consultas a las BDs.

Por otro lado, los usuarios de internet que consultan la página de resultados, al ingresar en su navegador descargan el *SPA* de la aplicación desde la red de AWS CloudFront. Al iniciar la aplicación, lo primero que hace es obtener la información pública a través del *endpoint* de la Lambda 2 de la arquitectura. Al finalizar, la aplicación procede a solicitar al *endpoint* de la Lambda 1 los registros de resultados. Luego transforma esa información para mostrarla correctamente en los gráficos de resultados.

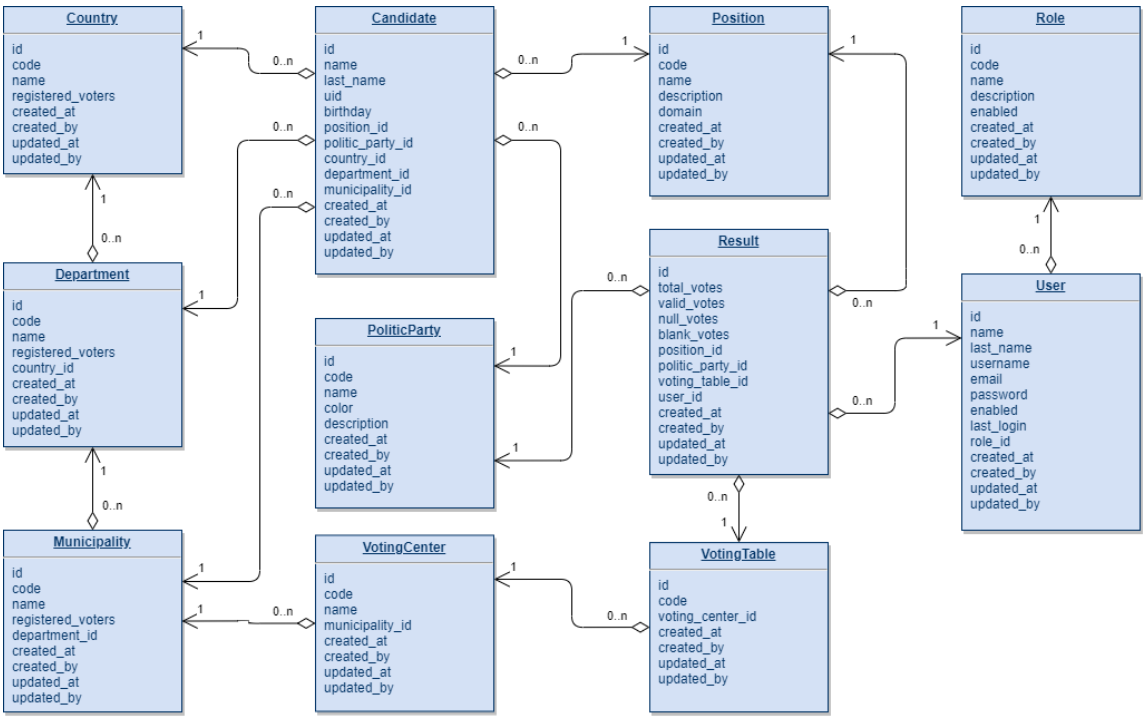
5.3. Diseño de la base de datos

La Figura 18 presenta el diagrama de entidad-relación de la BD completa del sistema de tabulación, donde se pueden visualizar todas las tablas que se crearon y los campos que se agregaron a cada una para almacenar la información de los votos. Para fines de auditoría, todos los registros guardan la fecha/usuario de creación y la fecha/usuario de la última actualización.

Se implementaron tablas para guardar los departamentos (*Department*), los municipios (*Municipality*) y los países (*Country*), que en este caso sería únicamente Guatemala. La importancia de los registros de estas tablas es que permiten validar la posición de un candidato, es decir, un candidato cuya posición es alcalde debe estar asociado obligatoriamente a un municipio. Otro dato importante que se guarda en estas tablas es el número de votantes registrados para todo el país, por cada departamento y por cada municipio

(*registered_voters*). Este valor se usa luego para calcular los porcentajes de participación y abstencionismo en la página de resultados.

Figura 18. **Diagrama de entidad-relación**



Fuente: elaboración propia, utilizando diagrams.net.

Se crearon tablas para administrar a los usuarios que tendrán acceso al sistema de tabulación (*User*) y el rol que corresponde a cada uno (*Role*), el cuál limita los permisos que el usuario tendrá dentro de la aplicación web. También se agregaron tablas para guardar a los partidos políticos (*PoliticParty*) y los candidatos (*Candidate*) por partido. Se almacena la información básica de un candidato y se asocia a un partido político, a una posición y, además ya sea a un país, a un departamento o a un municipio. Una posición (*Position*) representa un

cargo político, como por ejemplo presidente, diputado, alcalde, entre otros. Tanto un candidato de presidente como uno de vicepresidente estará asociado al país de Guatemala, mientras que todos los diputados estarán amarrados a un departamento, así como todos los alcaldes a un municipio.

Para manejar la logística de centros y mesas de votación se crearon las tablas *VotingCenter* y *VotingTable*. Un municipio puede tener varios centros de votación, así como un centro puede tener varias mesas de votación. Finalmente, la tabla más importante de la base de datos es la que se llama *Result* ya que es donde se almacenan los registros de resultados totalizados por cada mesa de votación. Los votos no se ingresan al sistema uno por uno, sino se crea un registro por el total de votos que obtuvo un partido político en una mesa de votación para cierta posición. Cuando se terminan de registrar los votos de todos los partidos que tienen candidatura para el municipio donde se encuentra la mesa de votación, se deben ingresar los registros que cierran la mesa (uno por cada posición). Un registro de cierre está asociado únicamente a una posición y guarda el total de votos nulos y votos en blanco. El sistema *backend*, al recibir un registro sin partido político, entiende que es un registro de cierre y contabiliza la mesa como procesada. Adicionalmente, los registros de resultados se asocian al usuario que los ingresó para fines de auditoría.

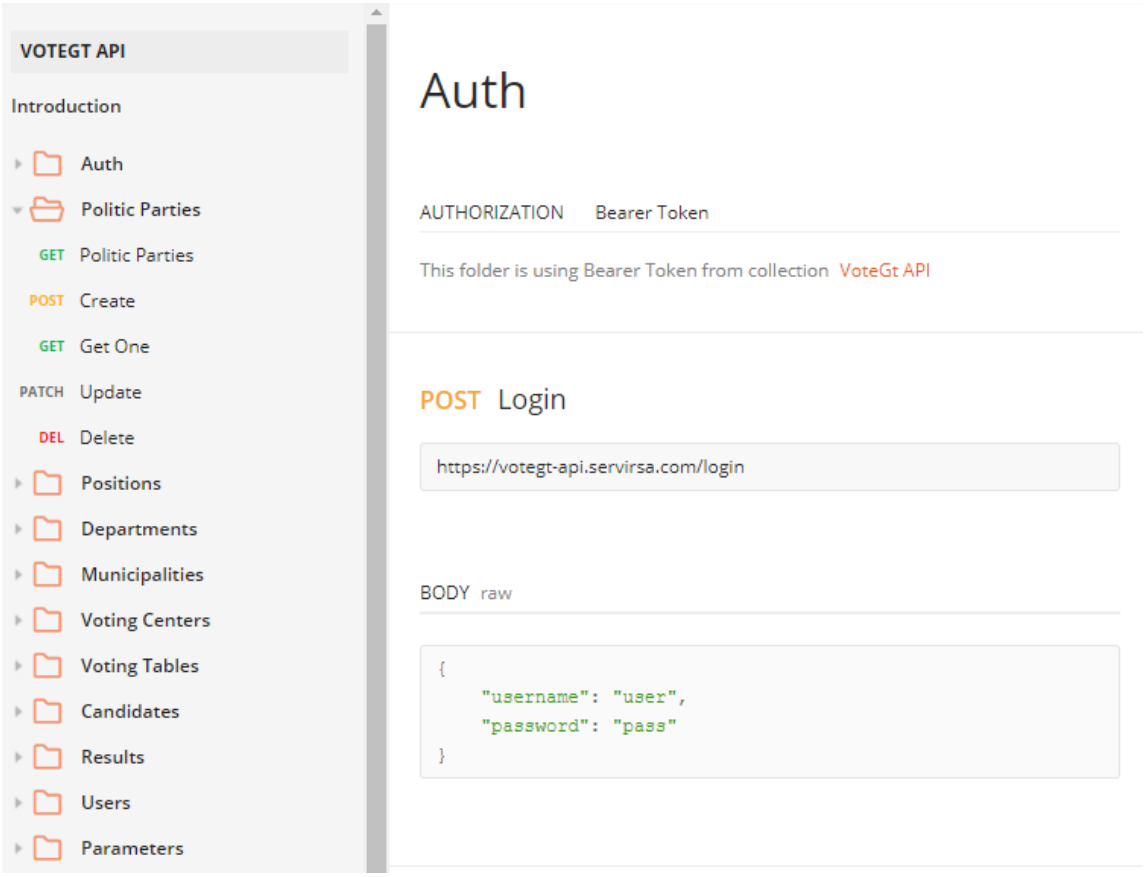
El RDBMS que se utilizó para la BD es MySQL y como ORM para el *backend* del sistema de tabulación se utilizó *Sequelize*. Este ORM permitió crear modelos que mapean a las tablas de la base de datos y luego mediante una instrucción de sincronización, se generó el esquema completo de la BD. Además, el ORM también permitió alimentar algunas tablas con data inicial para no tener que crearlos manualmente después.

5.4. Microservicios

El sistema completo se compone de tres microservicios en forma de APIs completamente desacopladas que son consumidas por las aplicaciones web y cada una cumple un propósito específico. El resultado final es un conjunto de *endpoints* que permiten consultar y manipular la información. En las siguientes figuras se muestran las pruebas realizadas con la aplicación de *Postman* para verificar el correcto funcionamiento de las APIs.

El microservicio para el sistema de tabulación que se muestra en la Figura 19, consiste en una API RESTful que expone los *endpoints* CRUD para cada una de las entidades que se muestran en la Figura 13. Además, maneja la lógica de autenticación de usuarios y la verificación de la autenticidad de cada solicitud que procesa mediante la evaluación de un token de sesión (*JWT*). Todos los *endpoints* tienen procesos de validación tanto a nivel de aplicación como a nivel de BD para mantener la consistencia de la información. Esta API se implementó con NodeJS usando el *framework* Express y para facilitar la manipulación de los datos, se utilizó el ORM Sequelize.

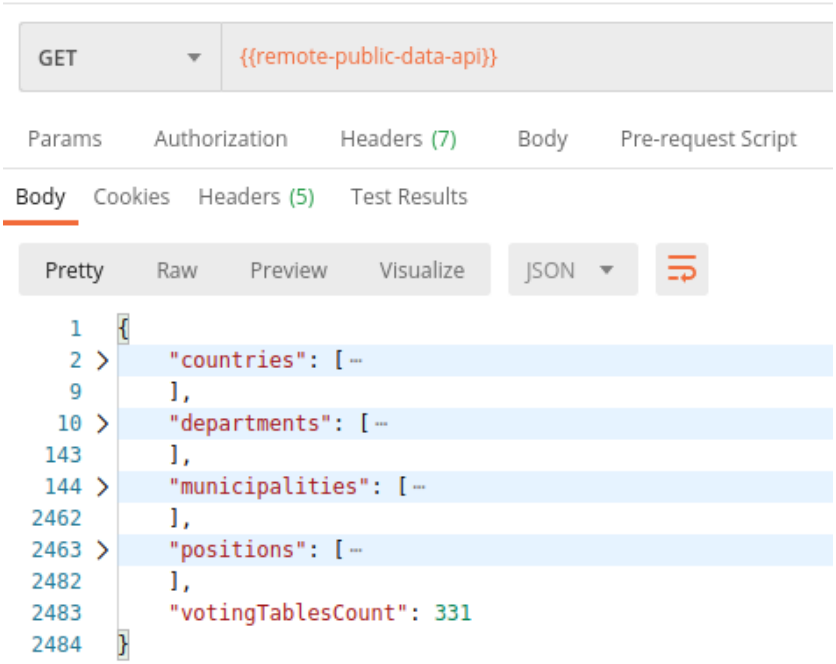
Figura 19. **API para el sistema de tabulación de resultados**



Fuente: elaboración propia, utilizando Postman.

El microservicio para consultar la información pública es un *endpoint* único expuesto por una función Lambda de AWS que devuelve los listados de países (en este caso, solo es Guatemala), departamentos, municipios y posiciones como se muestra en la Figura 20. Además, también devuelve el total de mesas de votación. Es una API pública y no tienen ningún método de autenticación.

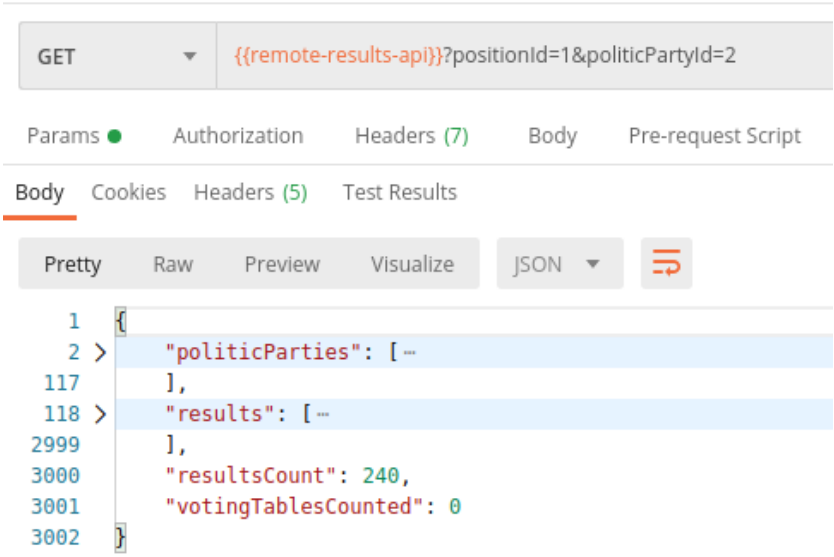
Figura 20. **API de consulta de la información pública**



Fuente: elaboración propia, utilizando Postman.

Por último, la Figura 21 muestra el resultado del microservicio para consultar los resultados, el cual también consiste en una función ejecutable en una Lambda que obtiene los registros de resultados de la BD y los devuelve sin aplicar ninguna transformación. La aplicación cliente se encarga de procesar y agrupar los datos para presentarla de forma gráfica. Devuelve también el listado de partidos políticos que aplican a los filtros enviados como parámetros, ya que no todos los partidos tienen candidatura para cierta posición en cierto municipio o departamento. Como datos adicionales para alimentar la página de resultados, se devuelve también la cantidad de mesas procesadas en la llave *votingTablesCounted*. Esta API tampoco tiene un método de autenticación porque expone es información pública.

Figura 21. **API de consulta de resultados**



Fuente: elaboración propia, utilizando Postman.

Las dos APIs públicas deben tener la estructura de una función que puede ejecutarse en una Lambda. Sin embargo, se trabajó primero en una versión que puede ejecutarse localmente antes de subirlo al servicio de AWS Lambda. Esta versión local, en vez de tener la lógica dentro de la función *handler* como lo requiere una Lambda, la expone en un *endpoint* mediante un servidor local de NodeJS. Para ello, también se utilizó el *framework* Express y la conexión a la BD se realizó de forma directa mediante la librería *mysqljs*.

5.5. Configuración de instancias de EC2

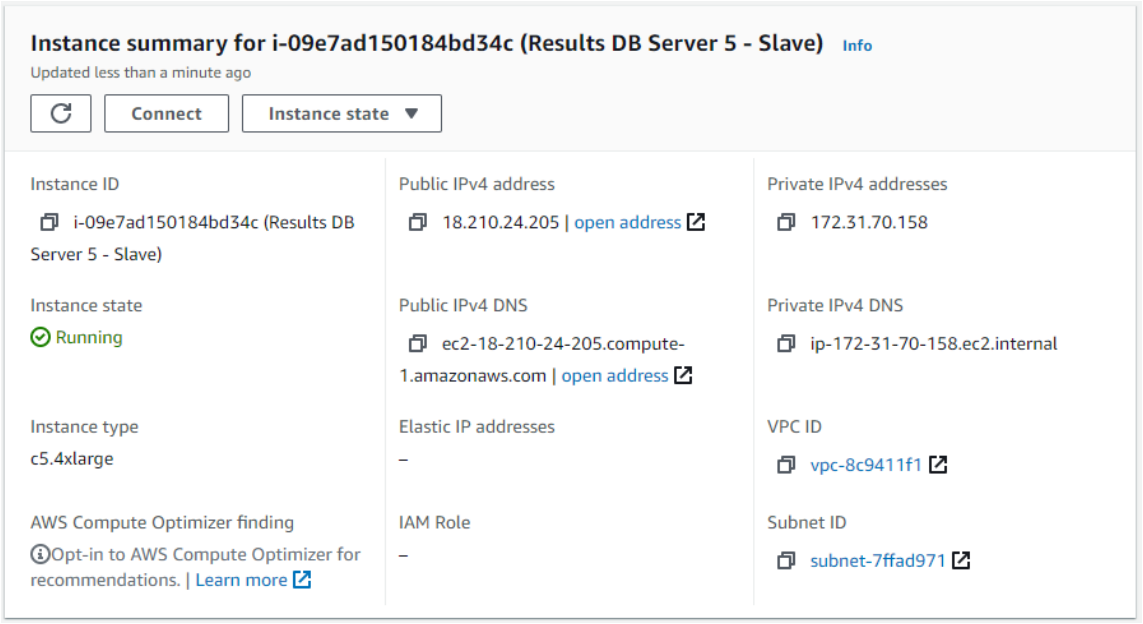
Antes de comenzar a describir los diferentes servicios que se provisionaron en AWS, se hace la aclaración de que se usó el idioma inglés en la interfaz de la consola de AWS ya que la versión en español presenta algunas incongruencias,

ya sea por mala traducción, porque se les olvidó traducir algunas palabras o porque simplemente el término no tiene una traducción correcta en español.

Como se observa en la Figura 17, la arquitectura que se propone requiere de cuatro instancias de EC2: una para la API del sistema de tabulación, una para la BD primaria y una por cada BD secundaria. Las instancias para la API y la BD primaria no van a recibir una alta carga de solicitudes, por lo que se eligió una de propósito general tipo "*t2.micro*" la cual ofrece un núcleo de procesador y 1 GB de RAM. En el caso de las instancias de las bases de datos secundarias, requieren de más recursos para soportar una alta carga de consultas y una alta concurrencia. Luego de varias iteraciones de pruebas, se descubrió que una instancia optimizada para procesamiento de tipo "*c5.4xlarge*" fue capaz de resolver el 99.95 % del total de solicitudes. Esta instancia ofrece 16 unidades virtuales de procesador y una RAM de 32 GB.

En el apéndice 1 se documenta el proceso completo para configurar cada instancia según su propósito. La Figura 22 muestra el resumen de la instancia donde se alojó la base de datos secundaria para consultar los resultados. Cuando se crea una instancia en AWS, se le asigna una IP pública y una IP privada. La privada es la IP que maneja AWS internamente dentro de la VPC y se puede utilizar entre los demás servicios para comunicarse internamente sin salir a la red pública.

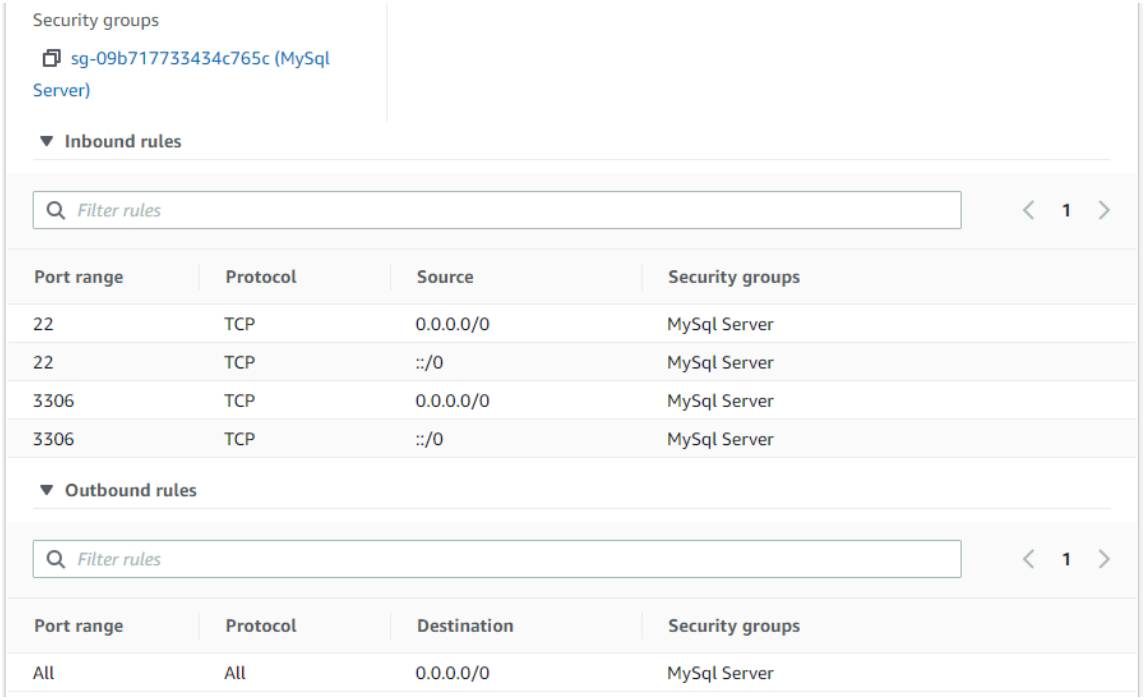
Figura 22. **Descripción de una instancia de EC2**



Fuente: AWS. (2021). *Consola de AWS EC2.*

Las instancias de EC2 necesitan ejecutarse bajo ciertas reglas de ruteo que limitan el tráfico de entrada (*Inbound rules*) y salida (*Outbound rules*), dichas reglas se configuran en un “Grupo de seguridad” (*Security group*). Como se observa en la Figura 23, la instancia está asociada al grupo “MySql Server” que permite el acceso mediante SSH (puerto 22) tanto en IPv4 como en IPv6, y el tráfico a través del puerto 3306 en IPv4 e IPv6, que es el puerto utilizado por defecto por MySQL. En las salidas se permite todo para tener acceso a internet y lograr comunicarse con las otras instancias. Como advertencia, esto podría representar una vulnerabilidad a nivel de seguridad. Se recomienda configurar las IPs y puertos específicos de destino y si se requiere acceso a internet, seguir las instrucciones de AWS para configurar una *Internet Gateway*.

Figura 23. **Grupo de seguridad de una instancia de EC2**



Fuente: AWS. (2021). *Consola de AWS EC2.*

5.6. Configuración de AWS Lambda

El servicio AWS Lambda permitió exponer las APIs para la consulta de la información pública y la consulta de los resultados. La Tabla III muestra la configuración de los parámetros principales de la Lambda de consulta de resultados, aunque esta misma configuración se puede utilizar para la otra Lambda. Los parámetros de conexión a la BD son variables de entorno y se pueden especificar en la sección "*Configuration* > *Environment variables*" de la función Lambda.

Tabla III. **Configuración de la Lambda para la consulta de resultados**

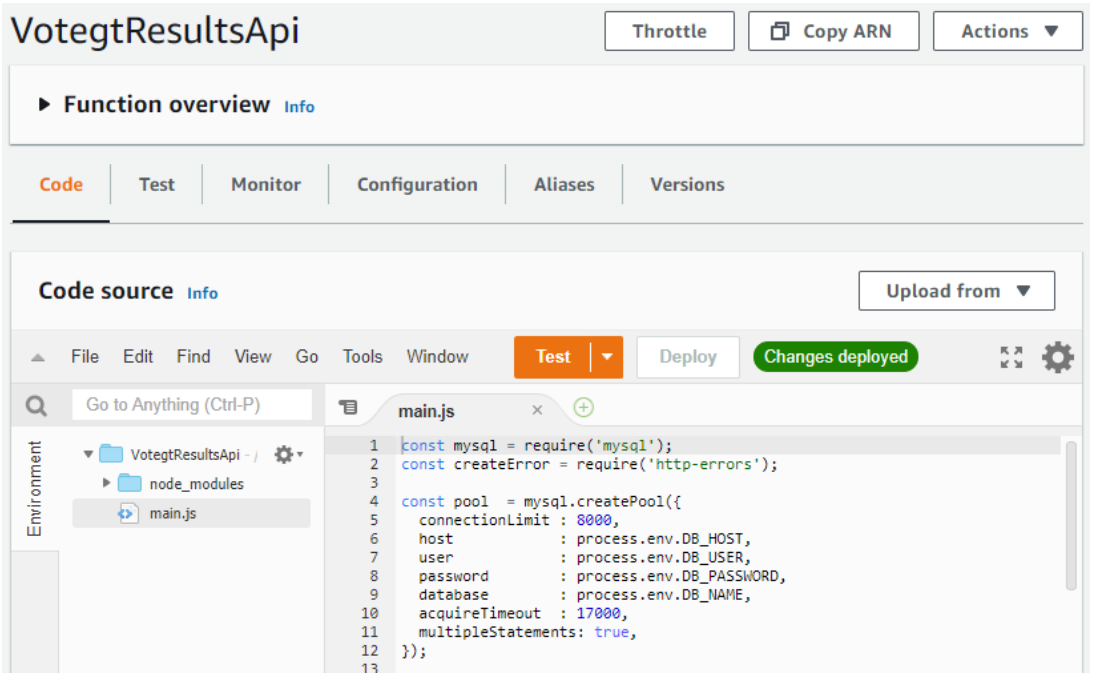
Parámetro	Valor	Descripción
Runtime	Node.js 12.x	Lenguaje de programación para implementar la función.
Handler	main.handler	Define el nombre del *script* y el nombre de la función que se va a ejecutar cuando se invoque la Lambda.
DB_HOST	172.31.25.132	Define la IP donde se encuentra la BD. Si la Lambda está en una VPC, se debe usar la IP privada.
DB_NAME	votegt_2021	Define el nombre de la BD desde donde se obtendrá la información.
DB_USER	MyUsername	Define el nombre del usuario para conectarse a la BD.
DB_PASSWORD	MySecretPass	Define la contraseña del usuario para conectarse a la BD.
Timeout	18 sec	Tiempo máximo que la Lambda debe esperar para resolver una solicitud. Si se pasa este tiempo, devuelve un error 500 con el mensaje "*Task timed out*".

Continuación tabla III.

Reserved concurrency	3890	Número máximo de solicitudes concurrentes que la Lambda va a poder procesar al mismo tiempo. Cuando se sobre pasa este límite, las solicitudes siguientes son rechazadas con un error de tipo 503.

Fuente: elaboración propia.

Figura 24. **Visualización de una Lambda en AWS**

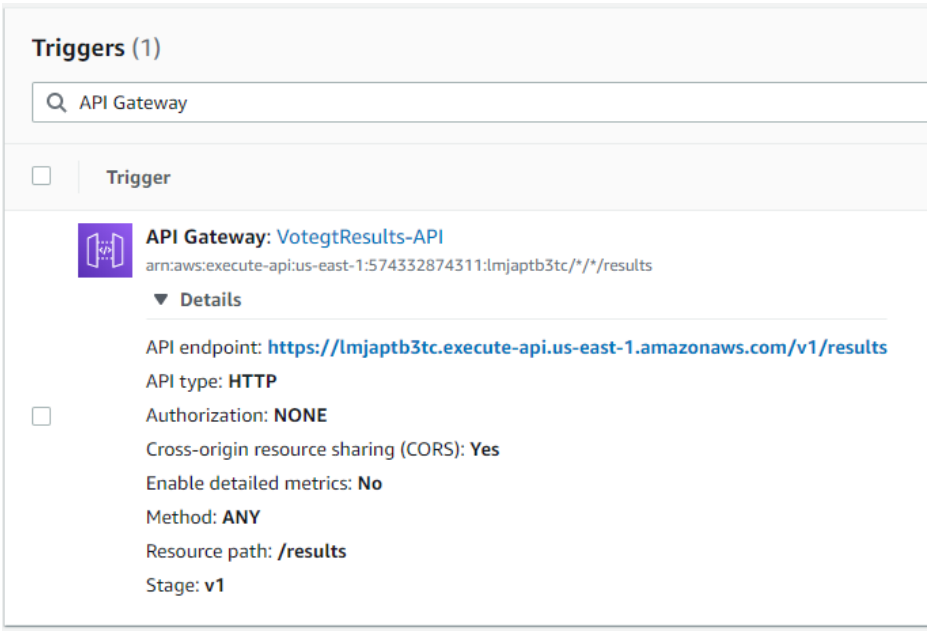


Fuente: AWS. (2021). *Consola de AWS Lambda.*

La Figura 24 visualiza una función Lambda en la consola de AWS y se muestran algunas líneas del código. También se puede observar que junto con el *script* que contiene la función a ejecutar, se debe subir también la carpeta de dependencias, que en el ambiente de NodeJS es la carpeta "*node_modules*". La forma más fácil de subir el código es creando un archivo comprimido ".zip" con el *script* principal y la carpeta "*node_modules*", luego usando la opción "*Upload from > .zip file*" en la pestaña "*Code*", se sube el archivo comprimido.

Una Lambda es simplemente una función con sus dependencias alojada en algún servidor de Amazon y no se ejecuta hasta que ocurra alguno de los eventos asociados a la función, que en el lenguaje de AWS se les llaman *Triggers*. Para cada Lambda de este proyecto, se configuró una API Gateway como *Trigger* de la función y esta consiste básicamente en un *endpoint* HTTP que puede llamarse desde cualquier aplicación web, desde cualquier aplicación *backend*, o incluso desde el mismo navegador. En la Figura 25 se puede visualizar el *endpoint* de la Lambda de resultados expuesto por la API Gateway y en el apéndice 2 se muestra el proceso de configuración.

Figura 25. **API Gateway como trigger para la Lambda de resultados**



Fuente: AWS. (2021). *Consola de AWS Lambda.*

5.7. Configuración de la BD en un RDBMS para una alta disponibilidad

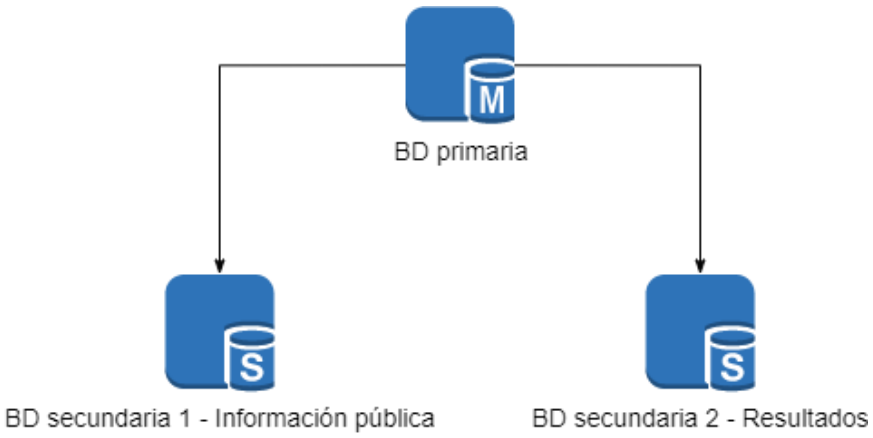
Como ya se mencionó en la presentación de la arquitectura del sistema completo, con el fin de mejorar la disponibilidad de la información a nivel de Base de Datos, se propone configurar una arquitectura de replicación *Master-Slave* para tener varias fuentes de datos y conectar cada microservicio con su propia base de datos.

5.7.1. Arquitectura de replicación Master-Slave

La arquitectura que se propone se muestra en la Figura 26 y se puede ver que consiste en un *cluster* de tres bases de datos en el que existe una base de

datos primaria y dos secundarias. Cualquier transacción que altera los datos en la BD primaria, se replica en las dos BDs secundarias conectadas al *cluster*. Cada BD secundaria posee un ID único y un usuario con los permisos necesarios para conectarse a la BD primaria y leer el archivo de bitácora de cambios (*binary log file*) para saber si necesita sincronizarse. Cada vez que se sincroniza, actualiza sus parámetros al último archivo leído y la posición leída. Las BDs secundarias se configuran como solo de lectura para evitar que se escriba información en ellas y resulte en una inconsistencia respecto a la BD primaria.

Figura 26. **Arquitectura de replicación Master-Slave en un RDBMS**



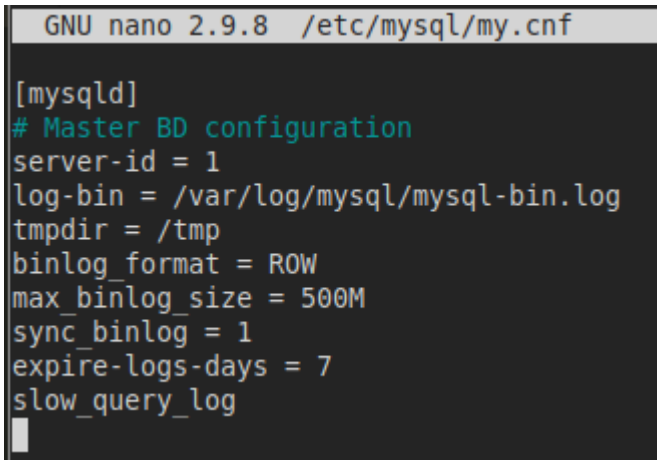
Fuente: elaboración propia, utilizando diagrams.net.

En una BD de tipo réplica se puede ejecutar el comando "*SHOW REPLICA STATUS*" para ver el estatus del nodo y visualizar si hay errores, ya sea de conexión con la BD maestra o de sincronización. Si todo está bien, las variables "*Slave_IO_Running*" y "*Slave_SQL_Running*" mostrarán el valor "*Yes*".

5.7.2. Configuración de la base de datos primaria (máster)

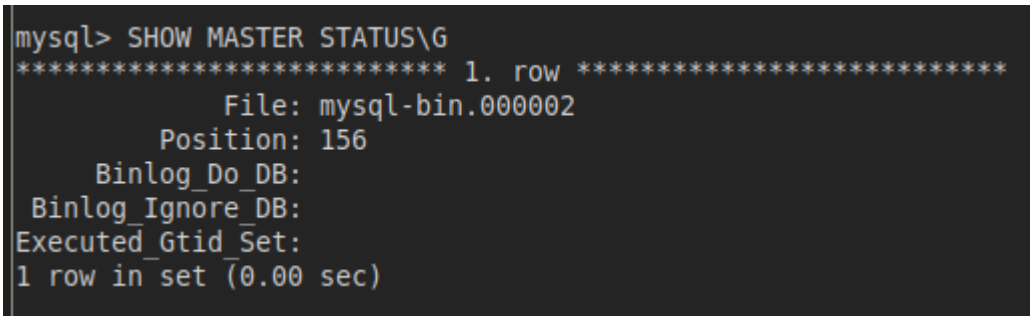
La Figura 27 muestra la configuración completa de la BD primaria que permite a los nodos de réplica conectarse y sincronizar sus tablas. Luego, La Figura 28 muestra el comando para verificar el estatus de la BD primaria. Los valores de las variables "*File*" y "*Position*" se usan para configurar el punto de inicio de sincronización de las BDs secundarias. En la BD primaria es necesario crear un usuario con permisos de réplica, dicho usuario será utilizado por las BDs secundarias para conectarse. El apéndice 3 muestra el proceso completo de configuración.

Figura 27. **Configuración de una BD primaria en MySQL**



Fuente: elaboración propia, con datos obtenidos de una instancia de EC2.

Figura 28. **Verificación del estado de la BD primaria**

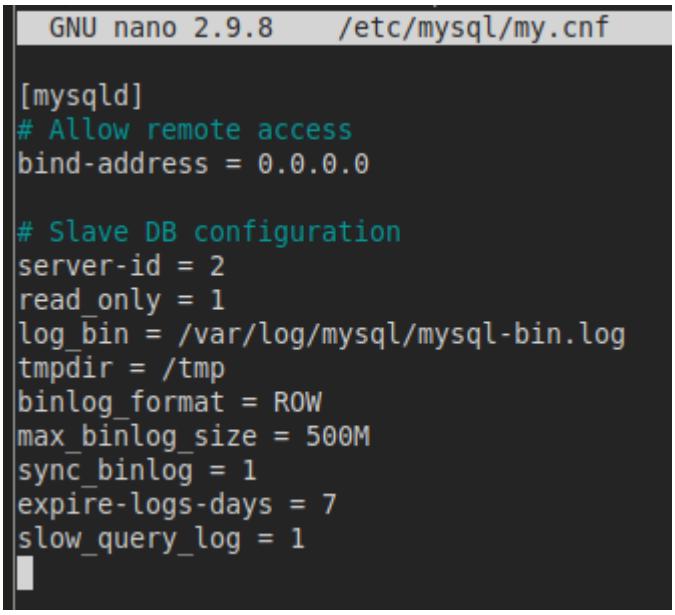


Fuente: elaboración propia, con datos obtenidos de una instancia de EC2.

5.7.3. Configuración de las bases de datos secundarias (slave)

Por otro lado, la Figura 29 muestra la configuración mínima de una base de datos secundaria para operar como un nodo de réplica. Para verificar el estatus de una BD secundaria, se puede utilizar el comando "*SHOW REPLICA STATUS\G*", tal como se observa en la Figura 30. De igual manera, el Apéndice 4 documenta el proceso completo para lograr la comunicación y sincronización entre las BDs primaria y secundaria.

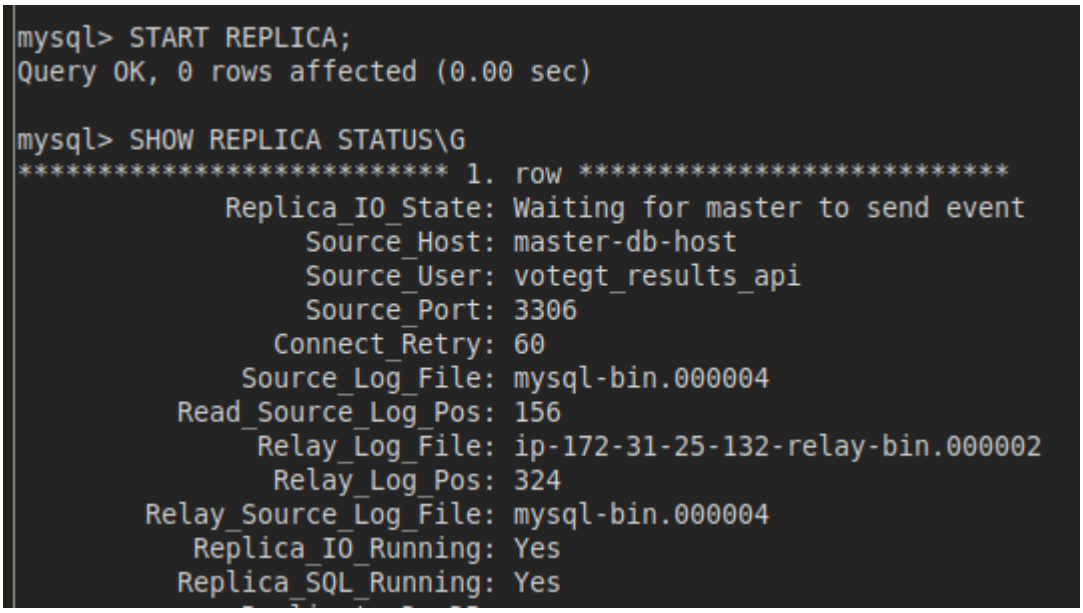
Figura 29. **Configuración de una BD de réplica en MySQL**



Fuente: elaboración propia, con datos obtenidos de una instancia de EC2.

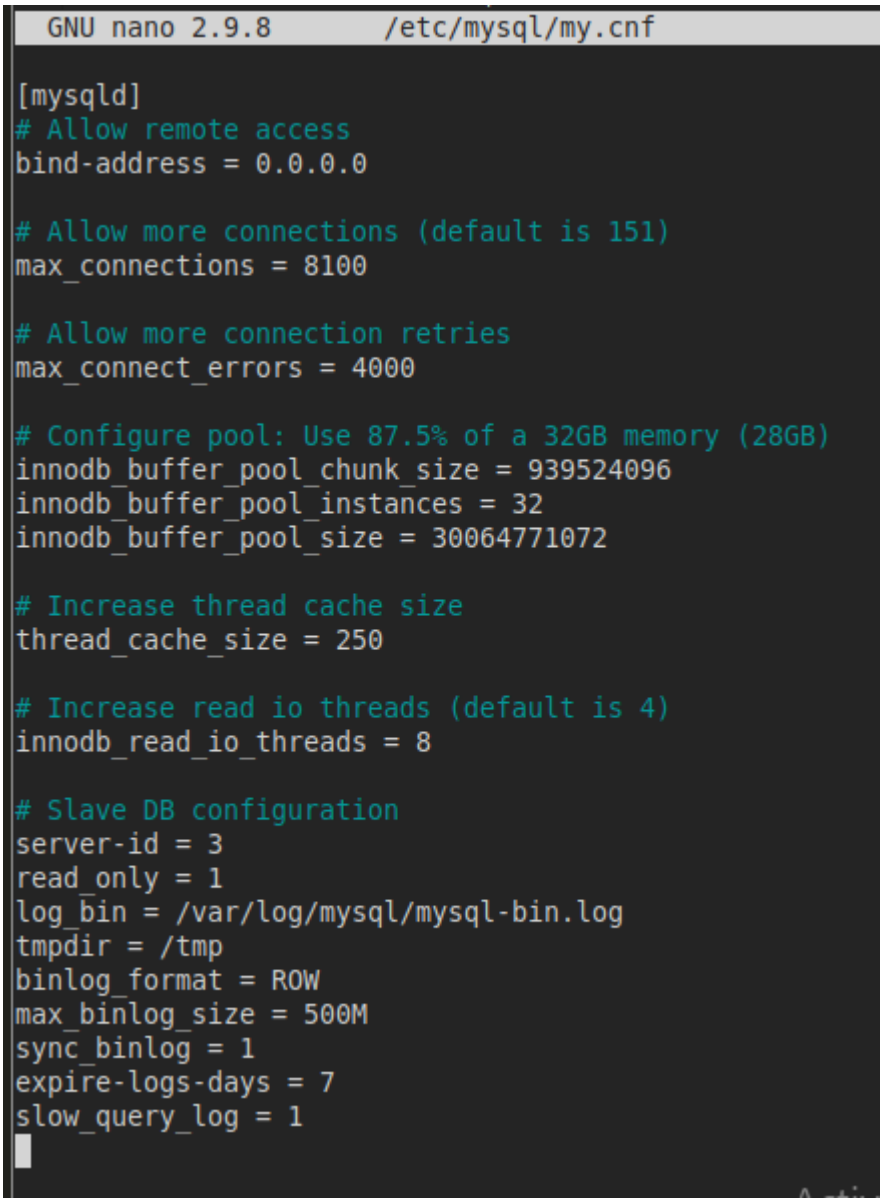
Aunque la configuración de la Figura 29 es suficiente para que una BD opere como un nodo de réplica, se requieren algunos ajustes adicionales para aprovechar al máximo los recursos disponibles de la instancia de la BD y optimizar MySQL para un alto rendimiento. La Figura 31 muestra la configuración final de la BD secundaria desde donde se consultan los resultados, optimizada para una instancia de tipo "*c5.4xlarge*". Luego de varias pruebas de carga, esta configuración es la que tuvo un mejor desempeño alcanzando un nivel de disponibilidad del 99.95%. En el Apéndice 9 se discute sobre cada uno de los parámetros configurados y su efecto sobre el rendimiento de la base de datos.

Figura 30. **Verificación del estado de una BD de réplica**



Fuente: elaboración propia, con datos obtenidos de una instancia de EC2.

Figura 31. **Configuración de una BD secundaria en MySQL para un alto rendimiento**



Fuente: elaboración propia, con datos obtenidos de una instancia de EC2.

5.8. Aplicaciones web

De acuerdo con el análisis inicial de los microservicios, se identificaron dos aplicaciones completamente independientes y con diferentes casos de uso, las cuales son el sistema de tabulación de resultados y la página de presentación de resultados en forma de gráficos estadísticos. Cada aplicación se desarrolló por separado utilizando el *framework* Angular versión 11 y para estilizar la interfaz de usuario, se utilizó el *framework* de CSS Bootstrap versión 4.

5.8.1. Sistema de tabulación de resultados

El sistema de tabulación de resultados permite la administración de las tablas que alimentan la página de resultados y lo más importante, permite el ingreso y/o modificación de los registros de resultados. Esta aplicación web está directamente conectada con la API del sistema de tabulación y utiliza el protocolo SSL/TLS durante el transporte de la información. El certificado de seguridad de la API se generó mediante *certbot* que es una herramienta de código libre (*open source*) que permite utilizar los certificados de *Let's Encrypt* para habilitar HTTPS en sitios web o APIs.

Un certificado de seguridad requiere de un nombre de dominio ya que se usa durante el proceso de verificación por los navegadores para validar el certificado. Para evitar el costo de adquirir un dominio, se alquiló un subdominio del dominio "servirsa.com" que pertenece a la empresa Servir SA. Desde la plataforma de GoDaddy se configuró un registro de "Tipo A" con el subdominio "*votegt-api*" apuntando a la IP pública de la instancia de EC2 donde está alojada la API del sistema de tabulación. Luego se configuró en la instancia un servidor

de Nginx que expone la API sobre HTTPS para que pueda ser accedido por el sistema de tabulación de forma segura. En el Apéndice 5 se encuentra la configuración completa del servidor de Nginx.

El sistema de tabulación de votos contempla tres roles, cada uno con permisos diferentes, los cuáles se describen en la Tabla IV.

Tabla IV. **Roles y permisos del sistema de tabulación**

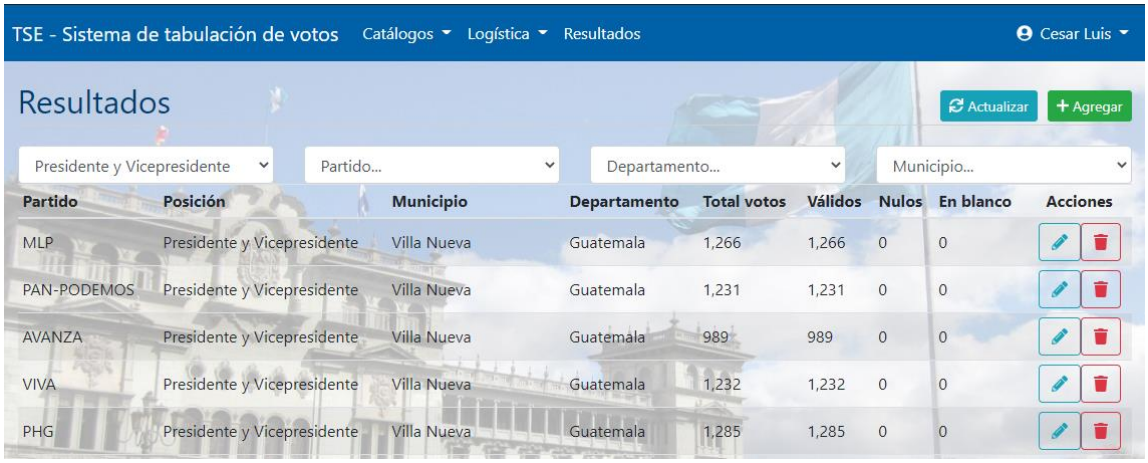
Nombre	Código	Permisos
Super Admin	SUPER_ADMIN	Puede manipular la información de todos los módulos del sistema.
Admin	ADMIN	Puede manipular la información de todos los módulos del sistema, a excepción del módulo de usuarios. Este módulo solo es gestionado por el Super Admin.
Tabulador	TABULATOR	Únicamente puede manipular la información del módulo de resultados.

Fuente: elaboración propia.

Por cada entidad del diagrama de modelo de dominio (Figura 13), se implementó un módulo en la aplicación para administrar los registros de dicha entidad. La Figura 32 muestra el módulo donde se ingresan los resultados y lista algunos registros de prueba que fueron generados por un proceso automático al inicializar la BD. El módulo permite agregar un nuevo registro, editar uno

existente, eliminar un registro y filtrar la consulta por posición, partido político, departamento y municipio.

Figura 32. **Módulo de resultados del sistema de tabulación**



Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

Figura 33. **Módulo de usuarios del sistema de tabulación**

Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

La Figura 33 muestra el módulo de gestión de usuarios y muestra las opciones de menú para llegar a los otros módulos agrupados en el menú de “Catálogos”. Este módulo permite al usuario “Super Admin” registrar los usuarios que van a tener acceso al sistema y asignarles un rol para limitar sus permisos.

En la Figura 34 se visualiza el módulo de “Departamentos y municipios” y entrando en el botón amarillo de cada departamento, se accede a la lista de municipios del departamento. Un dato importante de los departamentos y municipios es que guardan el número de votantes inscritos y este valor se usa para calcular el porcentaje de abstencionismo. Este módulo junto con el módulo de “Centros de votación” componen el menú de “Logística”.

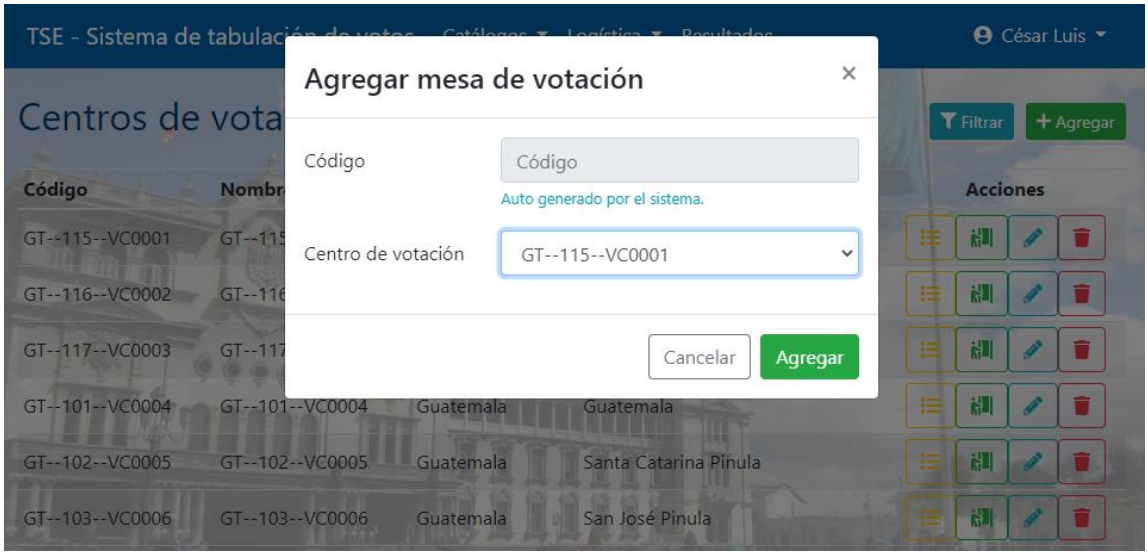
Figura 34. **Módulo de departamentos y municipios del sistema de tabulación**

Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

El sistema fue diseñado para permitir múltiples mesas en cada centro de votación y en la Figura 35 se muestra cómo registrar una nueva mesa a un centro

de votación existente. Al final, el único dato que necesita es un código auto generado por el sistema, pero el registro es importante ya que este es el que se asocia a los registros de resultados, es decir, un resultado debe pertenecer a una mesa de votación.

Figura 35. **Registro de mesa de votación en el sistema de tabulación**

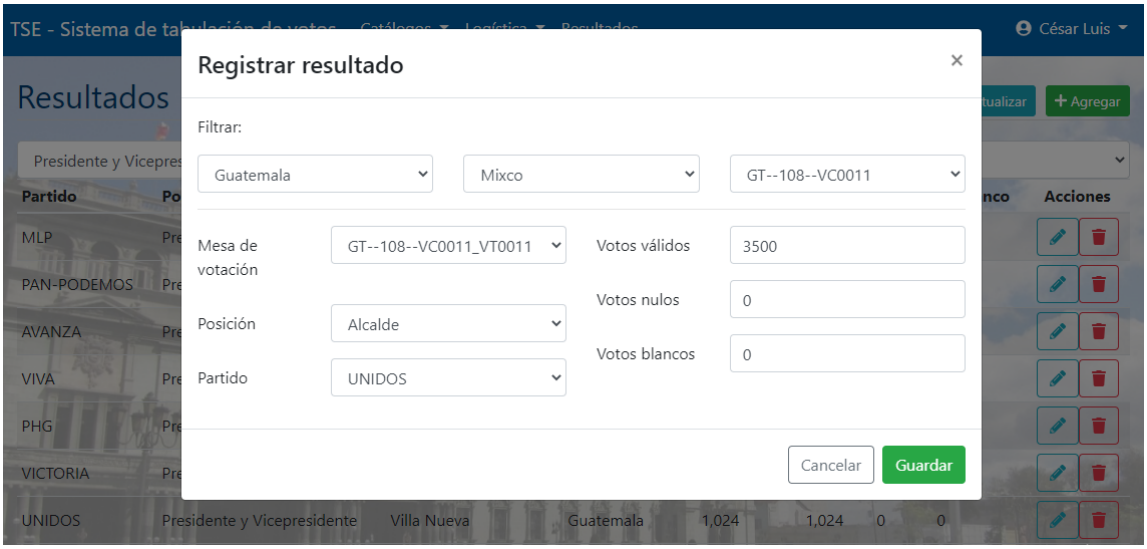


Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

Por último, en la f se puede observar el formulario que permite ingresar un registro de resultado y requiere que se especifique la mesa de votación, la posición, el partido político y la cantidad de votos válidos. Dado que hay cientos de mesas de votación, se debe primero filtrar por “Departamento”, luego por “Municipio” y finalmente por “Centro de votación”. Los partidos políticos que se va a listar solo serán aquellos que tienen candidatura para la posición seleccionada y para el departamento o municipio seleccionado (dependiendo si es para “alcalde” o “diputado”). El proceso para cerrar una mesa de votación es

creando un registro de resultado para la mesa donde únicamente se especifica la posición y el valor de los votos nulos y blancos (si hubiesen).

Figura 36. **Formulario para el ingreso de resultados en el sistema de tabulación**



Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

5.8.2. Página de presentación de resultados

A continuación, se muestran algunas pantallas de la aplicación web para la consulta de resultados mostrando datos de pruebas que se generaron automáticamente al inicializar la BD. Cabe mencionar que se tomó como base la página de resultados actual del Tribunal Supremo Electoral, tanto a nivel de interfaz como a nivel de arquitectura de la información. Inicialmente, la página muestra los resultados para “presidente y vicepresidente” a nivel nacional. Luego, se puede filtrar para las otras posiciones (“diputado” y “alcalde”), por partido

político, por departamento y por municipio como muestra la Figura 37. Cuando la posición seleccionada es “alcalde”, obligatoriamente se debe seleccionar un municipio. Lo mismo pasa con la posición “Diputado” que requiere la selección de un departamento.

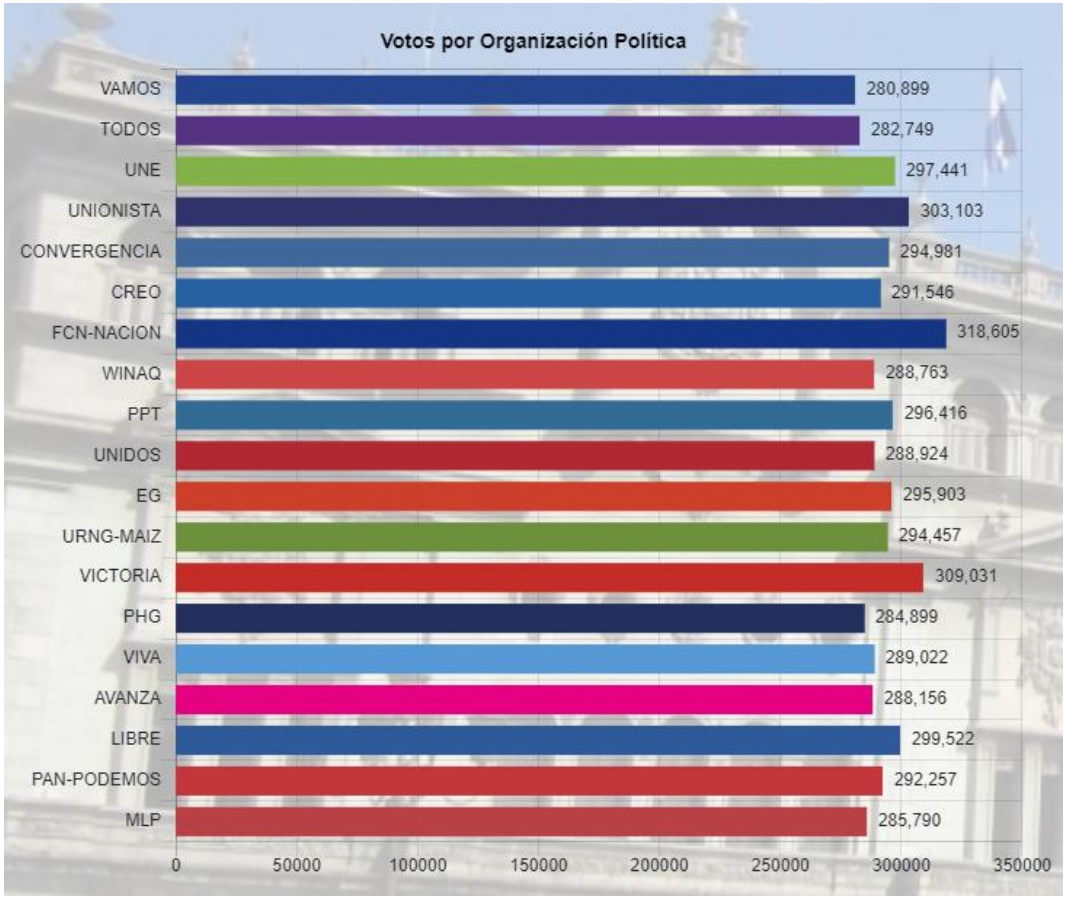
Figura 37. **Encabezado y filtros de la página de resultados**

Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

Esta aplicación web consume los dos microservicios públicos expuestos por las dos Lambdas. Cuando se carga la página por primera vez, la aplicación obtiene la información pública en una de las Lambdas (países, departamentos, municipios, posiciones y total de mesas). Al finalizar este proceso, solicita los registros de resultados a la otra Lambda y una vez los obtiene, procede a tabular los datos para agrupar por partido político y calcular los porcentajes de votos. Además, calcula los porcentajes de mesas procesadas, así como los porcentajes que alimentan el gráfico de abstencionismo.

Los resultados que se muestran en la Figura 38 fueron generados por un *script* que crea los registros de forma aleatoria dependiendo si un partido político tiene candidatura para cierta posición en cierto departamento o municipio.

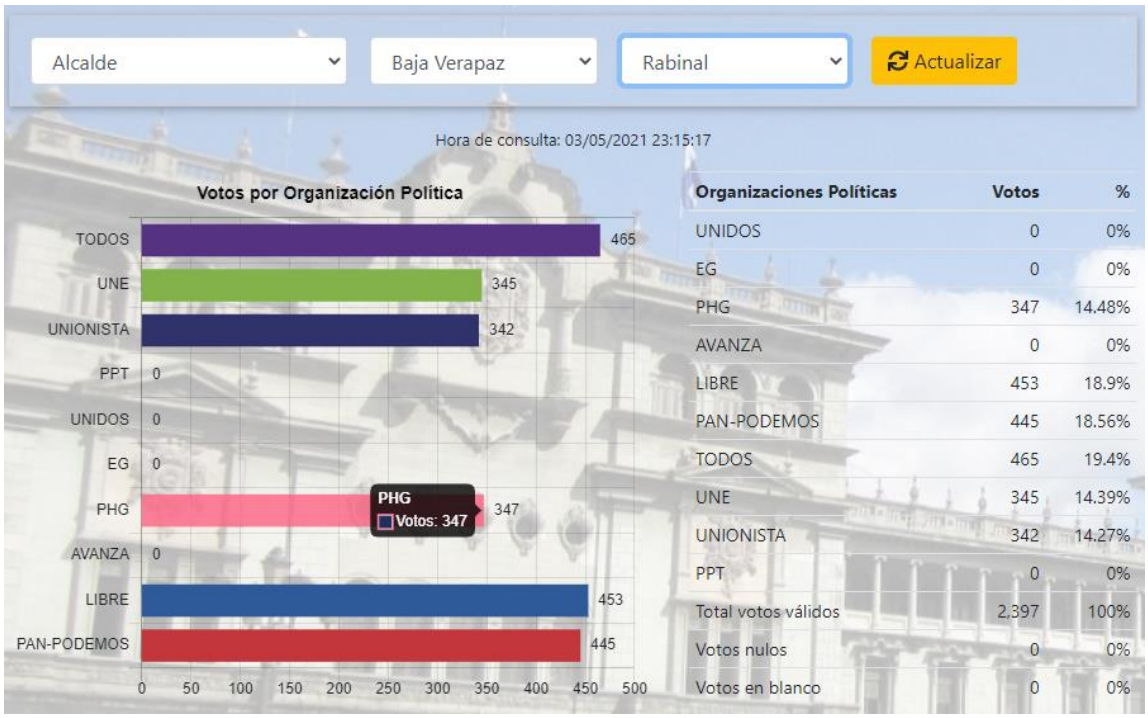
Figura 38. **Gráfico de resultados tabulados**



Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

El gráfico de la Figura 39 muestra los resultados filtrados por la posición de “alcalde” para el municipio de “Rabinal” del departamento de “Baja Verapaz”. Se puede observar que no se listan todos los partidos políticos de la Figura 38 ya que la gráfica toma en cuenta si un partido no tiene candidatura para la posición seleccionada en el departamento o municipio seleccionado.

Figura 39. **Resultados filtrados por la posición "alcalde"**



Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

La Figura 40 muestra los resultados tabulados en forma de tabla y el porcentaje de votos que le corresponde a cada partido político respecto del total de votos. La página de resultados también posee una tabla que muestra la información del total de mesas y cuántas de estas ya fueron procesadas, tal como se observa en la Figura 41.

Figura 40. **Resumen de resultados tabulados**

Organizaciones Políticas	Votos	%
VAMOS	280,899	5.03%
UNIDOS	288,924	5.18%
EG	295,903	5.3%
URNG-MAIZ	294,457	5.27%
VICTORIA	309,031	5.54%
PHG	284,899	5.1%
VIVA	289,022	5.18%
AVANZA	288,156	5.16%
LIBRE	299,522	5.37%
PAN-PODEMOS	292,257	5.24%
MLP	285,790	5.12%
TODOS	282,749	5.06%

Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

Figura 41. **Tabla de mesas procesadas**

Mesas Procesadas		
Mesas procesadas	0	0%
Mesas no procesadas	331	100%
Total mesas	331	100%

Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

Por último, también muestra un gráfico de pastel que indica el porcentaje de participación de los votantes como se observa en la Figura 42. Desde el sistema de tabulación se configura el total de votantes inscritos por cada departamento y

cada municipio. Usando este valor y el total de votos, se calculan los porcentajes de participación y abstencionismo.

Figura 42. **Gráfico de abstencionismo**

Fuente: elaboración propia, utilizando Angular 11 y Bootstrap CSS 4.

5.9. Rendimiento de la arquitectura de alta disponibilidad en AWS

Para probar el rendimiento de la arquitectura en AWS se realizaron varias pruebas con distintos escenarios de configuración, variando los parámetros de MySQL, las configuraciones de la Lambda y a medida que se alcanzaba la capacidad máxima de la instancia de la BD, se migraba a otra con más recursos. Este proceso de prueba y error se detalla en el apéndice 9, donde se analiza y discute el impacto que cada parámetro generó sobre la arquitectura, así como los errores que se encontraron y cómo estos fueron resueltos.

Luego de varias iteraciones de pruebas, se encontró que la configuración óptima para la arquitectura que logra resolver más del 99.9% de las solicitudes, es la que se detalla en la Tabla V. Con esta configuración se realizó la prueba final de integración de la Lambda de resultados y la API del sistema de tabulación. No fue posible realizar en paralelo la prueba de la Lambda para consultar la información pública ya que la cuenta de desarrollador de Artillery no permite ejecutar dos pruebas a la vez en AWS Fargate. Por otro lado, la prueba de la API del sistema de tabulación se ejecutó localmente ya que solo realiza una solicitud a la vez a cada 10 segundos.

Tabla V. **Configuración de la prueba final**

Descripción	2000 solicitudes a cada 10 segundos durante 5 minutos distribuidos entre 20 tareas en AWS Fargate.
Instancia EC2	
Tipo	c5.4xlarge
vCPUs	16
Memoria RAM	32GB
MySQL JS	
connectionLimit	8000
acquireTimeout	17000ms
Query timeout	17000ms
MySQL	
max_connections	8100
max_connect_errors	4000
innodb_buffer_pool_size	28GB => 30064771072 B

Continuación tabla V.

innodb_buffer_pool_chunk_size	0.875GB => 939524096 B
innodb_buffer_pool_instances	32
innodb_read_io_threads	8
thread_cache_size	250
Lambda	
Timeout	18s
Reserved concurreny	3890
Artillery	
Timeout	60s

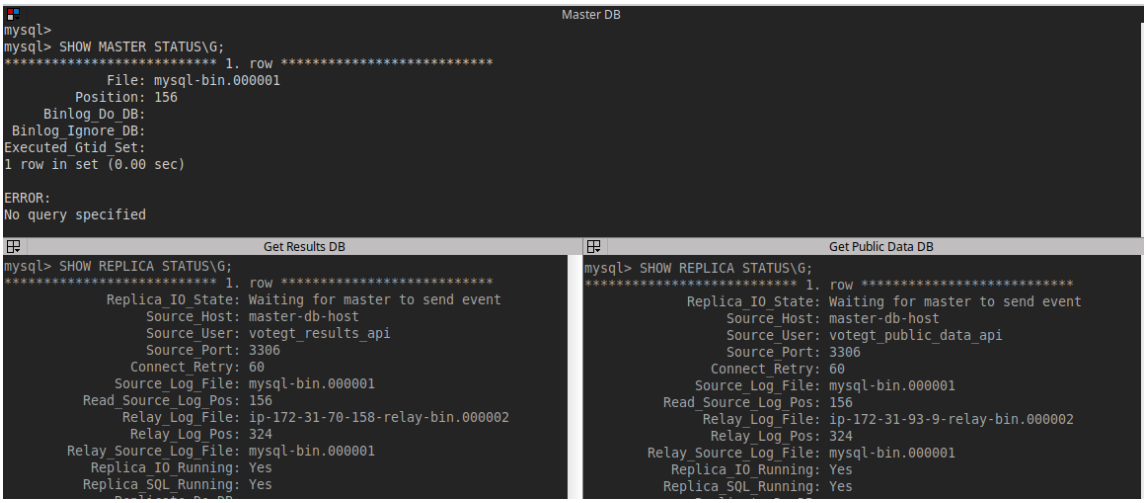
Fuente: elaboración propia.

A continuación, se presentan los resultados de una iteración de pruebas de carga con esta configuración, así como imágenes de cómo cambió el estado de las bases de datos. El procedimiento de Artillery que pone a prueba la API del sistema de tabulación, envía una solicitud para crear un registro de resultado a cada 10 segundos durante 5 minutos, lo que resulta en 30 registros nuevos ingresados a la BD primaria. Estos *INSERT's* se propagan eventualmente a las bases de datos secundarias, lo cual se corroboró mediante una consulta de tipo *SELECT*.

Para comprobar que las bases de datos secundarias están configuradas correctamente y se encuentran conectadas a la BD primaria, se usa el comando "*SHOW REPLICA STATUS\G*" y se confirma que el valor de las dos variables "*Replica_IO_Running*" y "*Replica_SQL_Running*" muestren el valor "*Yes*", tal como se aprecia en la Figura 43. Estas variables indican si el *thread* que verifica

si hubo cambios en la BD primaria y el *thread* que aplica los cambios para sincronizar, respectivamente, se encuentran activos y en ejecución. En caso contrario, el error se mostrará en la variable "*Last_Error*".

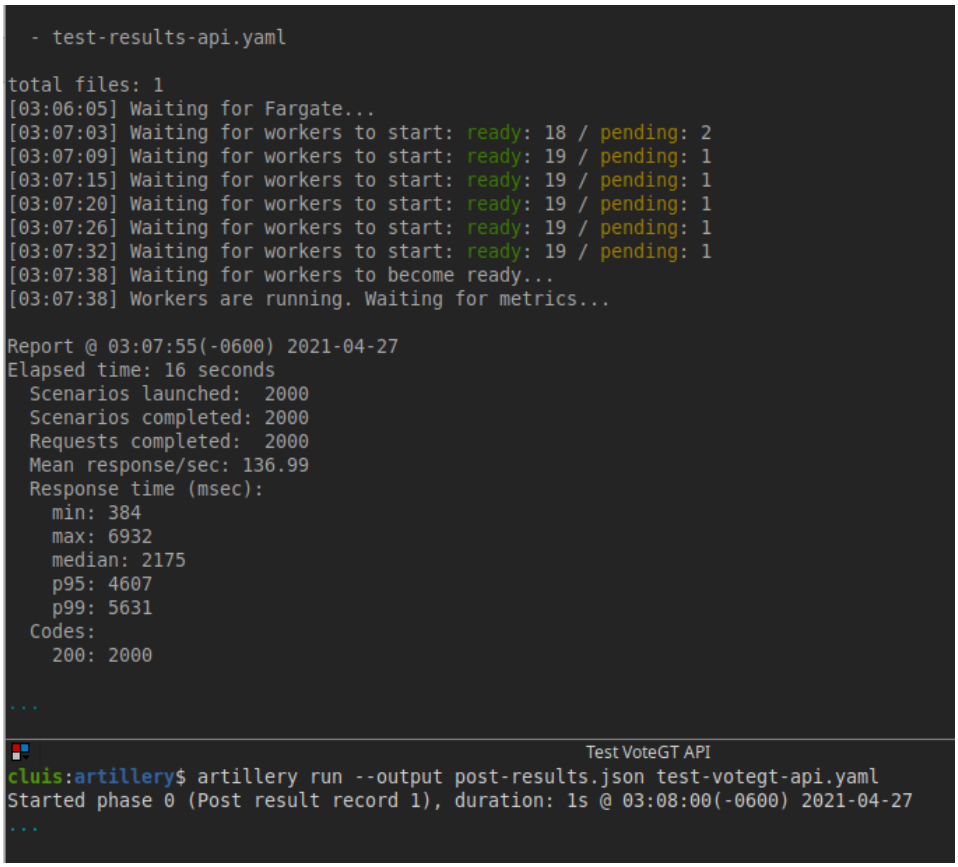
Figura 43. **Estatus de replicación Master-Slave**



Fuente: elaboración propia, con datos obtenidos de tres instancias de EC2.

La Figura 44 muestra la inicialización de las dos pruebas de Artillery que se ejecutaron en forma paralela. La prueba en AWS Fargate tarda un tiempo en iniciar ya que primero debe inicializar el *cluster* de ECS e instanciar las 20 tareas que van a ejecutar las pruebas. Una vez las tareas están listas para ejecutarse, se obtiene el mensaje "*Workers are runing. Waiting for metrics…*". Cuando apareció este mensaje, se ejecutó la prueba de Artillery que valida la disponibilidad de la API del sistema de tabulación.

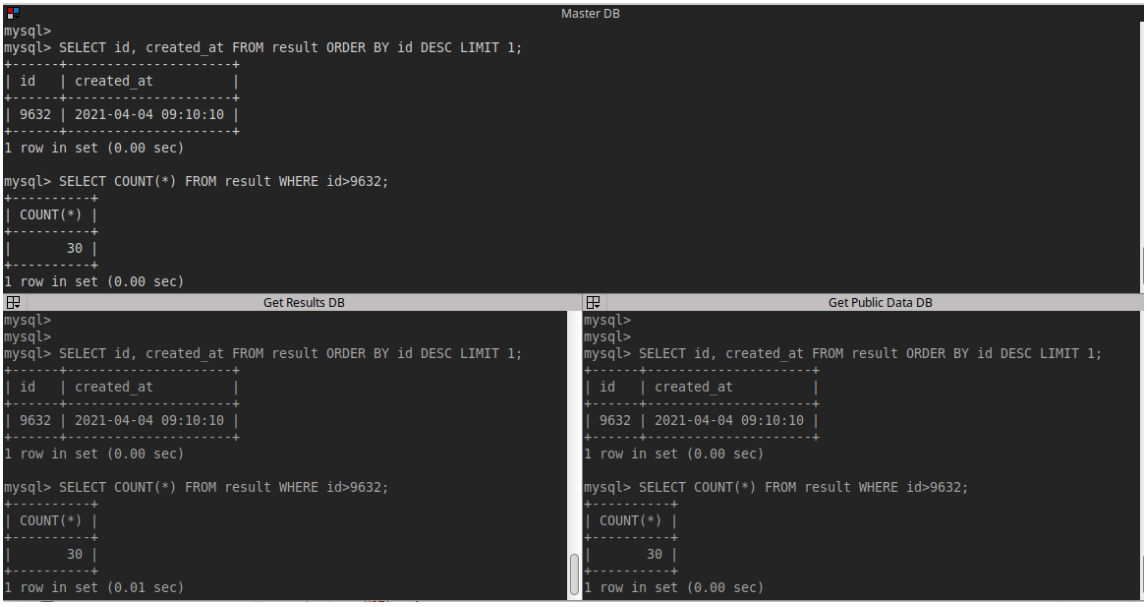
Figura 44. **Inicialización de pruebas de Artillery para ejecutar en paralelo**



Fuente: elaboración propia, con datos obtenidos de una ventana de comandos local.

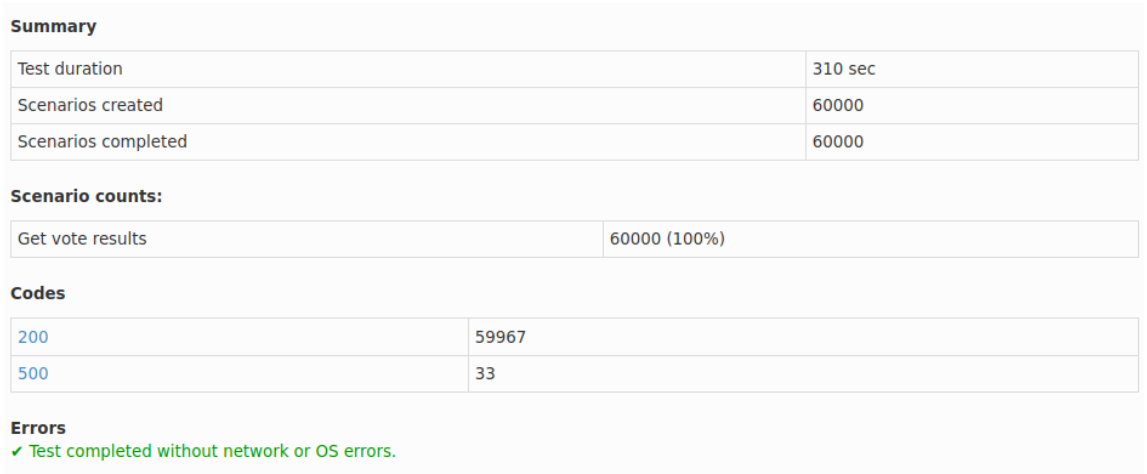
Las consultas de MySQL que se observan en la Figura 45 muestran el cambio que sufrieron las BDs luego de completar las pruebas contra la API del sistema de tabulación. El primer "*SELECT*" en las 3 terminales demuestra que el último registro de resultado en la BD tiene el ID 9632. Al finalizar las pruebas, se ejecutó el segundo "*SELECT*" y este confirma que existen 30 registros nuevos después del ID 9632, tanto en la BD primaria como en las secundarias.

Figura 45. **Estatus de las bases de datos antes y después de las pruebas**



Fuente: elaboración propia, con datos obtenidos de tres instancias de EC2.

Figura 46. **Lambda de resultados - resultados de Artillery**



Summary

Test duration	310 sec
Scenarios created	60000
Scenarios completed	60000

Scenario counts:

Get vote results	60000 (100%)

Codes

200	59967
500	33

Errors

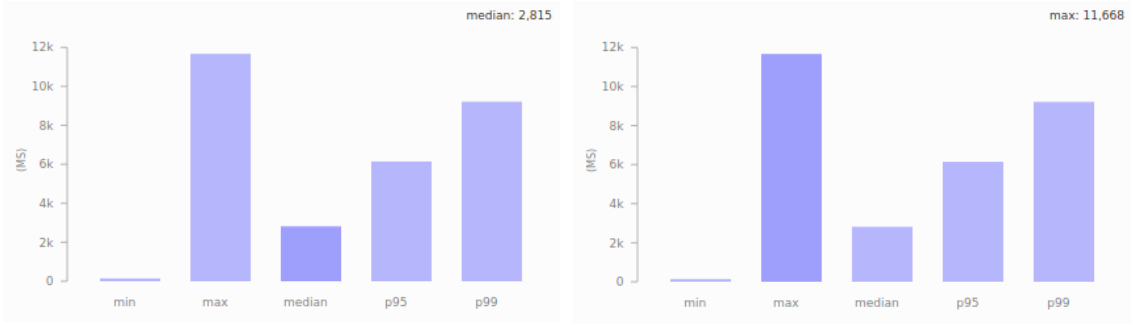
✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

El reporte de Artillery sobre la prueba de la Lambda de resultados (Figura 46) indica que se resolvieron 59,967 solicitudes exitosamente de los 60,000 que se programaron, pero 33 finalizaron con un error de tipo 500, que normalmente indican *timeouts* en la Lambda. Los resultados de Artillery también indican que la prueba completa tardó 310 segundos, lo cual corresponde a los 5 minutos de todas las fases más el tiempo de inicialización. Sin embargo, este tiempo no representa el tiempo en que se completaron todas las solicitudes.

Otro de los resultados que genera Artillery consiste en un gráfico sobre la distribución de la latencia de las solicitudes realizadas. La Figura 47 muestra el mismo gráfico dos veces para visualizar la mediana y el valor máximo de latencia y como se puede observar, el tiempo máximo que una solicitud tomó en completarse fue de 11.7 segundos. Además, también indica que el 95 % de las solicitudes tardaron menos de ~7 segundos en completarse, mientras un 99 % lo hizo en menos de ~10 segundos.

Figura 47. **Lambda de resultados - distribución de latencia**

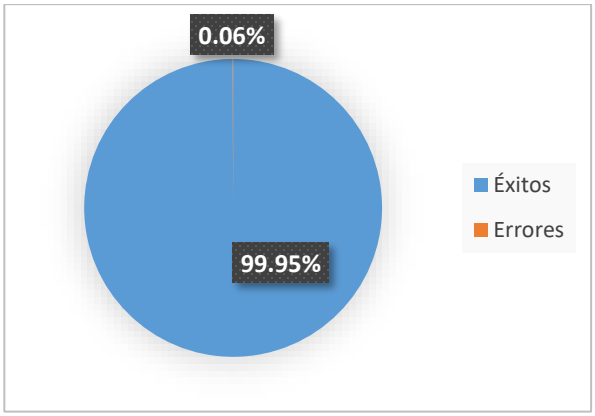


Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

Con base en la cantidad de éxitos y errores reportados por Artillery, se calculó que el nivel de disponibilidad de la Lambda es del 99.95 %, tal como lo muestra el gráfico de pastel de la Figura 48.

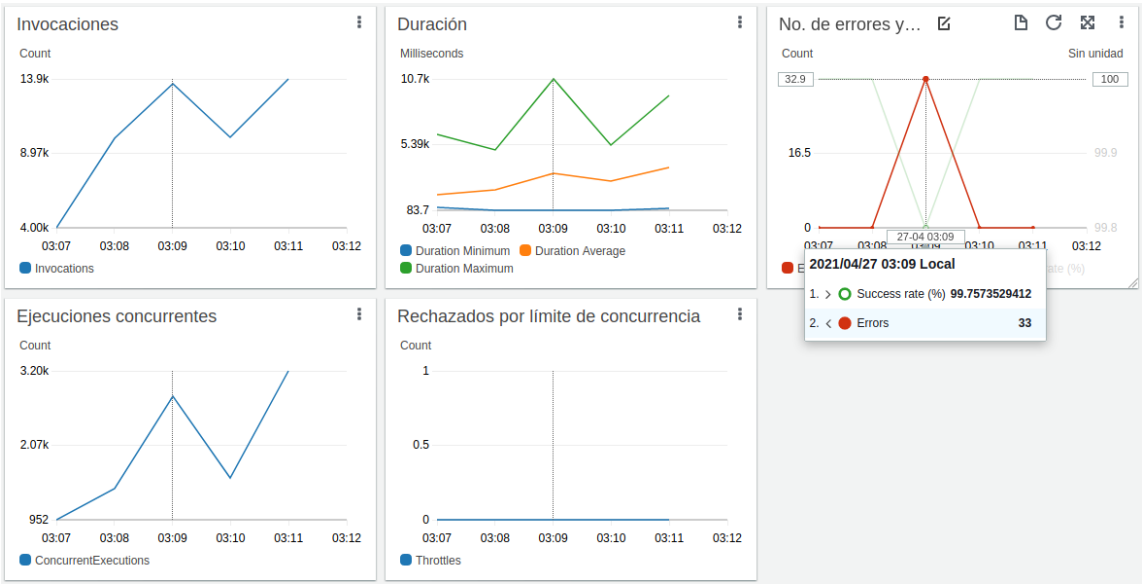
La Figura 49 muestra las métricas de la Lambda de resultados durante la ejecución de las pruebas de carga y como puede observarse, el número de invocaciones alcanzo un pico de ~13,900, mientras la cantidad de ejecuciones concurrentes llegó hasta ~3,200. El gráfico de duración de las ejecuciones confirma los resultados de Artillery, donde se puede observar que el promedio se mantuvo bajo 5 segundos, mientras el máximo fue alrededor de 10.7 segundos. Por su parte, el gráfico de errores muestra la misma cantidad de errores reportados por Artillery.

Figura 48. **Lambda de resultados - gráfico de disponibilidad**



Fuente: elaboración propia.

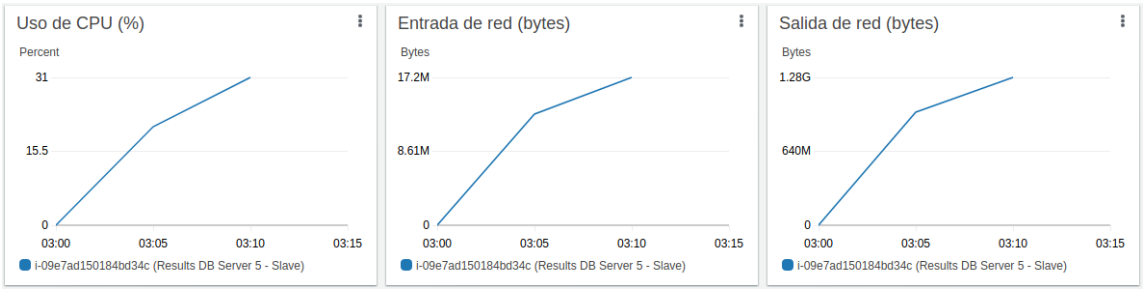
Figura 49. **Lambda de resultados - métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch..

Los gráficos de monitoreo de la instancia de EC2 de la base de datos de resultados (Figura 50), indican que el porcentaje de uso del procesador alcanzó hasta un 31 %. Además, el tráfico de las solicitudes entrantes llegó hasta 17.2 MB, pero el tráfico de salida fue mucho más alcanzando un máximo de 1.28 GB.

Figura 50. **Lambda de resultados - monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch..

Por otro lado, las 30 solicitudes de la prueba contra la API del sistema de tabulación se completaron correctamente como se observa en los resultados de Artillery de la Figura 51, lo cual indica un nivel de disponibilidad del 100 %. Además, de acuerdo con los gráficos de la Figura 52 el tiempo máximo que demoró una solicitud fue de 140 milisegundos.

Figura 51. **API del sistema de tabulación - resultados de Artillery**

Summary

Test duration	310 sec
Scenarios created	30
Scenarios completed	30

Scenario counts:

Post result record	30 (100%)

Codes

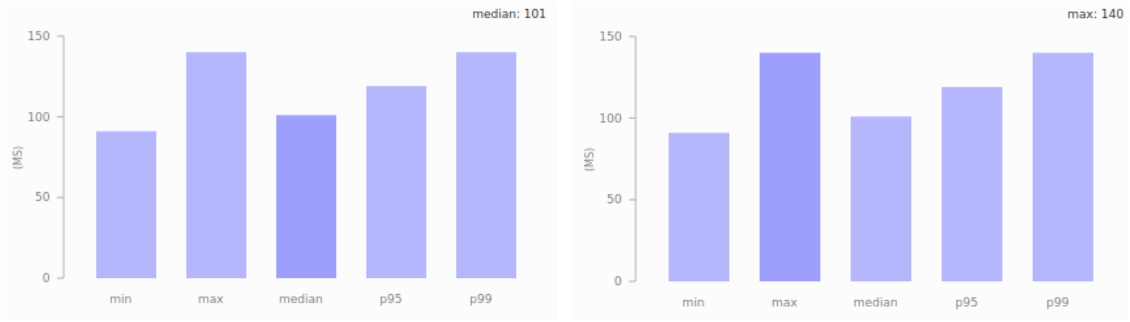
200	30

Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

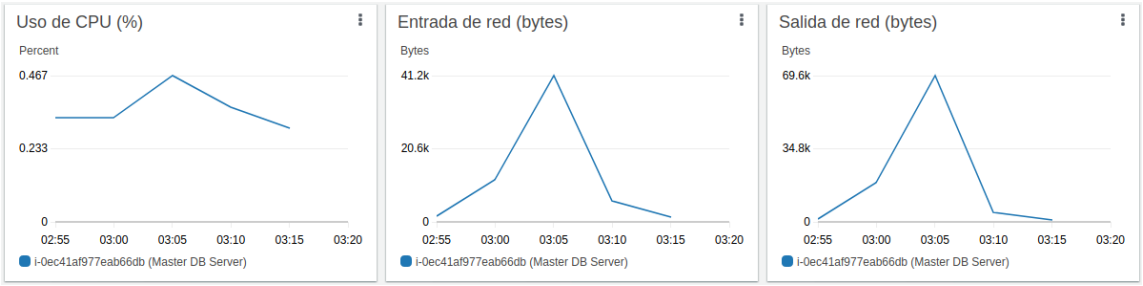
Figura 52. **API del sistema de tabulación - distribución de latencia**



Fuente: elaboración propia, uyilizando comando de Artillery para generar reportes.

Por último, la Figura 53 muestra los gráficos de monitoreo de la instancia de EC2 de la API, e indican un incremento leve en el uso del procesador y un nivel de tráfico de red también mínimo.

Figura 53. **API del sistema de tabulación - monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch..

6. DISCUSIÓN DE RESULTADOS

A continuación, se realiza una valoración del trabajo realizado, los resultados obtenidos y las lecciones aprendidas luego de completar el proyecto. Se hace un análisis del desempeño de los componentes de la arquitectura, las complicaciones de su implementación y se resaltan las configuraciones más importantes que ayudaron a mejorar su rendimiento. Por último, se analizan los resultados que se obtuvieron con las pruebas de carga y se hace un juicio constructivo sobre las decisiones tomadas y cómo estos repercutieron sobre los resultados finales.

La arquitectura que se propuso tiene por objeto resolver el problema de disponibilidad que mostró la página de resultados del TSE durante las elecciones de Guatemala del año 2019. Sin embargo, había que tomar en cuenta que la página de resultados mostraba información que se estaba ingresando en tiempo real desde otro sistema y la dependencia o el nivel de acoplamiento entre dichos sistemas, podía influir sobre el rendimiento de cada sistema individual. Por esta razón, se consideraron dos aplicaciones del TSE para prototipar una arquitectura orientada a microservicios de alta disponibilidad utilizando principalmente los servicios Lambda y EC2 de AWS.

Específicamente, se propuso implementar una arquitectura en AWS que soporte tanto el sistema de tabulación de votos como la página de presentación resultados, de manera que esta última pudiera soportar 2000 usuarios concurrentes sin afectar la disponibilidad del primero. Luego para demostrar esta

capacidad, se realizó una prueba de carga de 2000 solicitudes a cada 10 segundos durante 5 minutos hacia el microservicio que devuelve los resultados y de forma paralela, se ejecutó otra prueba para demostrar la disponibilidad del sistema de tabulación enviando solicitudes que crean registros de resultados.

Para implementar un sistema que soporte una alta concurrencia de usuarios, la arquitectura física debe encajar correctamente con la arquitectura lógica. Una arquitectura de alta disponibilidad requiere atención en todas las capas de transmisión de la información: base de datos, arquitectura lógica, arquitectura física, a nivel de aplicación cliente, entre otros. Si se descuida alguna capa, se puede convertir en un cuello de botella y degrada la operación de las otras.

6.1. Análisis y diseño de microservicios

El primer paso fue analizar el problema completo y tratar de mapear cada necesidad del sistema a un componente de software. Para ello, se aplicó parcialmente el método de *Domain Driven Design* (DDD) para identificar las entidades del dominio completo y los contextos aislados (Evans, 2003). Gracias al resultado de este proceso y al diagrama de casos de uso se identificaron los módulos que componen el sistema completo y, en consecuencia, los microservicios aislados que agrupan módulos con una alta cohesión. La arquitectura lógica que se propuso en la Figura 16 podría no corresponder a una arquitectura de microservicios si se toma en cuenta el tamaño de la API del sistema de tabulación. Definitivamente, se podría desglosar en microservicios más pequeños pero debido al nivel de dependencia que existe entre las entidades, se decidió dejar en un solo microservicio.

Sin embargo, la arquitectura completa sí se adhiere a un patrón de microservicios porque cada una de las tres APIs que se definió, puede operar de manera independiente y cuenta con su propia BD, es decir, son APIs totalmente desacopladas y la disponibilidad de una no afecta la disponibilidad de las otras dos. La razón de definir un microservicio para la consulta de los resultados y otro para la consulta de la información pública es que los datos que provee cada uno tienen una frecuencia de uso diferente. Los resultados se actualizan constantemente, pero la información pública (departamentos, municipios, partidos políticos, posiciones, número de mesas) será la misma durante todo el proceso de tabulación.

6.2. Modularización de las aplicaciones web

Las aplicaciones web se implementaron como *Single Page Application* (SPA) con el objetivo de separar la capa de presentación de la capa de manipulación de los datos. No hace mucho tiempo, la lógica de presentación de los sistemas web se manejaba completamente del lado del servidor (*server rendering*), de modo que cuando se navega entre las URLs del sistema, se descarga toda la página junto con la data. Hoy en día, los sistemas web modernos se implementan como SPAs y los servidores de *backend* solo se encargan de exponer APIs que permiten consultar y manipular la información. Siguiendo este paradigma, la página de presentación de resultados consume dos APIs públicas expuestas por dos funciones de AWS Lambda, mientras la aplicación para la tabulación de los resultados consume una API RESTful con autenticación, expuesta por una instancia de EC2.

Para reducir la cantidad de consultas de lectura a la BD, se diseñó la página de resultados para que cargue la información pública una sola vez y luego el usuario use el botón de “Actualizar” que se muestra en la Figura 37 para obtener únicamente los resultados. De esta manera, se obtiene únicamente la información que se necesita y se evita sobre cargar las BDs con consultas repetitivas.

6.3. Disponibilidad a nivel de base de datos

Para mejorar la disponibilidad de la información desde la capa de base de datos, se propuso una arquitectura de replicación *Master-Slave*, de manera que las operaciones de escritura se aplican únicamente en la BD primaria, mientras las consultas de solo lectura se realizan sobre las BDs secundarias. Los casos de uso del sistema de tabulación y presentación de resultados se ajustan perfectamente a esta arquitectura ya que la página de resultados no hace más que consultar la información y el único que necesita realizar operaciones de escritura es el sistema de tabulación. Gracias al análisis de microservicios que se realizó al inicio, se determinó que eran necesarias dos BDs secundarias para los dos microservicios de la página de resultados.

En arquitecturas más complejas con muchas bases de datos secundarias, es importante tomar en cuenta que el proceso de sincronización puede tener un impacto sobre el desempeño de la BD primaria ya que debe manejar una conexión por cada réplica y crear un archivo de bitácora de eventos (*binary log file*) por cada una. Asimismo, la consistencia eventual entre la BD primaria y las secundarias tomará más tiempo.

En cuanto al sistema de base de datos considerado para soportar la arquitectura de alta disponibilidad en operaciones de lectura, se decidió utilizar una base de datos relacional por la naturaleza y el volumen de la información que se va a manipular. Los registros de los resultados de las elecciones de Guatemala son datos de alto grado de sensibilidad y confidencialidad, por lo que es importante que el sistema de base de datos garantice la consistencia, integridad y durabilidad de la información. Este requerimiento es precisamente el lema y objetivo de las bases de datos relacionales que garantizan que las transacciones se realizan de forma atómica, consistente, aislada y durable (*ACID properties*), a diferencia de las BDs NoSQL que no satisfacen totalmente las características ACID porque están pensadas para almacenar estructuras de datos complejas o para sistemas de bases de datos distribuidas. La estructura de tablas del sistema de tabulación de votos es simple y el volumen de información que se va a almacenar es mínimo.

Dentro de la familia de RDBMS, se optó por utilizar MySQL ya que este manejador ofrece soluciones nativas para configurar arquitecturas de réplica en cualquier topología, y sobre todo posee una amplia documentación sobre la arquitectura de replicación *Master-Slave*. Por otro lado, MySQL es altamente configurable y permite al usuario optimizar su rendimiento de acuerdo con las necesidades de la aplicación. Es conocido también entre la bases de datos relacionales por su alto desempeño en operaciones de tipo lectura. No se optó por un sistema más sofisticado como SQL Server o el servicio de Amazon RDS que garantiza una alta disponibilidad debido a su costo elevado.

6.4. Microservicios

Para permitir la consulta de la información pública y los resultados tabulados desde la página de resultados, se implementaron dos funciones que pueden ejecutarse en una Lambda de AWS. Cada función se conecta a su BD secundaria correspondiente, realiza la consulta de la información que le interesa y la devuelve como tal sin aplicar ninguna transformación. Las Lambdas de los microservicios se configuraron en la misma VPC y misma subred de las instancias para usar su IP privada, mejorando así la velocidad de acceso a los datos.

El microservicio del sistema de tabulación consiste en una API RESTful autenticada que provee *endpoints* para crear, leer, actualizar y eliminar los registros de la BD primaria. Para desplegar esta API en la nube, se utilizó una instancia de EC2 con recursos mínimos ya que va a recibir pocas solicitudes de ingreso de resultados y como consecuencia, la BD primaria tampoco tendrá una alta carga de consultas. Una instancia de tipo "*t2.micro*", la cual ofrece una unidad de procesador y 1 GB de RAM, fue suficiente para resolver las 30 solicitudes que se generaron durante las pruebas. De la misma manera, la BD primaria se configuró a una instancia de tipo "*t2.micro*" y no tuvo ningún problema en procesar las 30 consultas de inserción.

6.5. Uso de AWS Lambda para una alta concurrencia

El servicio de AWS Lambda está diseñado para soportar altas cargas de trabajo y cada vez que hay una invocación de la función, se crea una instancia totalmente independiente con su propio ambiente y recursos de ejecución. Sin

embargo, tiene un límite de concurrencia que debe configurase tomando en cuenta las necesidades de la aplicación que va a consumir la función. Este parámetro se llama “Concurrencia reservada” o “*Reserved concurrency*” en inglés. Cuando las ejecuciones concurrentes de la Lambda alcanzan el límite de concurrencia reservada, las siguientes invocaciones son rechazadas con un error de tipo 503 (*Service Unavailable*).

Por defecto, cada cuenta en AWS tiene un límite de 1000 ejecuciones concurrentes por cada zona de disponibilidad (*Availability Zone o AZ*). La arquitectura propuesta debe soportar 2000 usuarios concurrentes y al tener dos Lambdas, la concurrencia esperada es de 4000. Para ello, se realizó una solicitud de ampliación de este parámetro directamente a AWS (*quota increase request*), la cual fue aprobada con éxito en alrededor de 4 días. El equipo de AWS Lambda requiere que se exponga por qué se quiere aumentar el límite de concurrencia, ellos lo evalúan y verifican si su infraestructura cuenta con la disponibilidad.

Las Lambdas permiten gestionar la alta concurrencia de solicitudes HTTP, mientras las BDs secundarias deben ser capaces de soportar la alta carga de consultas de lectura. La eficiencia de las dos capas de la arquitectura garantiza el nivel de disponibilidad de la página de resultados.

No fue posible realizar la prueba de estrés contra la Lambda para consultar la información pública, ya que la licencia de desarrollador que se logró obtener de Artillery solo permite ejecutar una prueba a la vez en AWS Fargate. Sin embargo, las dos Lambdas son completamente independientes y cada una se conecta a su propia BD, de manera que la configuración de la Lambda de

resultados se puede también aplicar a la Lambda que provee la información pública. Lo mismo aplica para las bases de datos secundarias.

Por otro lado, la página de resultados fue diseñada para que la información pública solo se consulte cuando se carga la página por primera vez, por lo que se espera un menor número de solicitudes en la Lambda de la información pública que en la Lambda de resultados. Sin embargo, en el peor de los casos el usuario podría ignorar el botón “Actualizar” de la página y simplemente recarga la página completa cada vez que quiere actualizar los resultados. Para cubrir este escenario, se recomienda utilizar la misma configuración para las dos Lambdas y las dos BDs secundarias.

El servicio de AWS Lambda dispone de una configuración que permite limitar el tiempo de ejecución de la función y cuando se alcanza este tiempo y la función aún no ha retornado una respuesta, AWS Lambda cancela su ejecución y devuelve un error de tipo 500 (*Internal Server Error*). A este tiempo se le llama *timeout* de la función Lambda. Este *timeout* tiene dos funciones importantes: 1) evitar ocupar una instancia de la función por mucho tiempo cuando un servicio con el que interactúa no responde, y 2) sabiendo que el servicio de AWS Lambda se paga por tiempo de uso, conviene evitar ejecuciones prolongadas para no elevar los costos.

En la prueba final de la Lambda de resultados que generó una disponibilidad del 99.95 %, fue necesario aumentar el *timeout* hasta 18 segundos y la concurrencia reservada hasta 3890 como se mostró en la Tabla V. La razón es que conforme se generaban más solicitudes, algunas no se lograban resolver en los 10 segundos de lapso entre cada iteración, por lo que eventualmente, el

número de ejecuciones concurrentes en la Lambda sobre pasaba el límite de 2000 configurado inicialmente, resultando en errores de tipo 503 para Artillery. Por esta razón, se incrementó la concurrencia reservada hasta 3890 y no se usó el total de 4000 disponibles porque AWS Lambda requiere que se dejen libres 100 concurrencias para que las otras Lambdas (o futuras Lambdas) puedan operar. La Lambda para obtener la información pública se dejó con las 10 unidades restantes.

Adicionalmente, existe otra configuración importante de las Lambdas que vale la pena mencionar. Cuando el requerimiento consiste únicamente en soportar una alta concurrencia de solicitudes, aumentar el límite de la concurrencia reservada y configurar correctamente el *timeout* de la Lambda es suficiente. Sin embargo, cuando el caso de uso requiere que el tiempo de respuesta sea lo más rápido posible, es necesario considerar otra configuración de la concurrencia de la función Lambda que se llama “concurrencia provisionada” (*provissioned concurrency*). Al momento de invocar una función Lambda, AWS crea una instancia de la función inicializando un contenedor con sus propios recursos para procesar la solicitud y el tiempo que tarda en levantar dicho contenedor puede tener un impacto negativo sobre el tiempo de respuesta. Para mitigar este problema, AWS implementó la capacidad de provisionar una cierta cantidad de instancias de la función, de manera que la función Lambda pueda responder casi de forma instantánea cuando hay una invocación.

La concurrencia provisionada mejora significativamente el tiempo de respuesta, sin embargo, eleva por mucho los costos ya que AWS cobra por la cantidad de instancias provisionadas y el tiempo que se encuentran activas, desde que se provisiona hasta que se deshabilita el servicio. No obstante, es

bueno tener en cuenta esta característica cuando se implementan sistemas que exigen un nivel mínimo de latencia.

6.6. Rendimiento de instancias EC2

Antes de hablar sobre el rendimiento de las instancias de EC2, vale la pena hacer una aclaración en cuanto a los núcleos físicos de procesador que una instancia posee. De acuerdo con la documentación de Amazon Elastic Compute Cloud, a excepción de las instancias de tipo T2, una vCPU es un hilo de procesamiento (*trhead*) de un núcleo físico de CPU y un núcleo puede tener hasta 2 hilos (Amazon Web Services, Inc., 2021). Esto significa que una instancia que dice tener 16 vCPUs, realmente tiene 8 núcleos físicos de procesador, pero tiene la capacidad de ejecutar hasta 16 procesos en paralelo.

Otro detalle importante acerca de las instancias de EC2 es que su IP pública cambia cada vez que son detenidas (apagadas), mientras que su IP privada permanece igual. Esto permitió crear nombres de dominio internos para referenciar a las otras instancias (en el archivo “/etc/hosts” en los sistemas Linux) y usar estos nombres para los permisos de los usuarios que se configuraron en las bases de datos. Como se muestra en el Apéndice 3, es necesario configurar en la BD primaria los usuarios que usan las BDs secundarias para conectarse y por seguridad, se habilita el acceso únicamente a la IP de la instancia de la BD. Usando las IPs privadas, evitó tener que reconfigurar los usuarios cada vez que se inician las instancias. Por otro lado, usar las IPs privadas mejora la latencia de la conexión ya que la comunicación se realiza en la red interna de AWS. Físicamente, las máquinas virtuales se encuentran en el mismo *data center* y usan la red interna para comunicarse en vez de salir a la red pública de internet.

Para encontrar el tipo de instancia de EC2 y la configuración que debe tener la BD para soportar 2000 solicitudes concurrentes, se tuvieron que realizar varias pruebas de carga variando la configuración de MySQL y cuando se alcanzaba la capacidad máxima de una instancia y aún no se había alcanzado una disponibilidad de al menos del 99.9 %, se migraba a una instancia con más recursos y se repetían las pruebas. En el Apéndice 9 se detalla el proceso de prueba y error que se realizó para encontrar la instancia de EC2 y la configuración óptima de MySQL que garantiza una disponibilidad de al menos del 99.9 % en la Lambda de resultados cuando recibe una concurrencia de 2000 solicitudes a cada 10 segundos.

Luego de varias iteraciones de pruebas, se encontró que la instancia de tipo "*c5.4xlarge*" fue capaz de resolver el 99.95 % de las solicitudes generadas. Esta instancia posee 16 vCPUs y una RAM de 32GB. Se utilizó esta instancia para obtener los resultados finales, sin embargo, se exploró un poco más allá probando con la siguiente instancia de la familia "c5" que es la "*c5.9xlarge*", la cual ofrece 36 vCPUs y 72GB de memoria RAM. Como se muestra en el Escenario 15 del apéndice 9, con esta instancia se logró un nivel de disponibilidad del 100 % y a la vez, se redujo el tiempo promedio que tardó la arquitectura para resolver cada solicitud.

6.7. Optimización de MySQL para una alta concurrencia

Por otro lado, fue necesario ajustar los parámetros de MySQL para una alta concurrencia, aprovechando al máximo los recursos de la instancia de EC2. MySQL viene con una configuración por defecto que, en la mayoría de los casos de uso y aplicaciones empresariales, es totalmente funcional y suficiente. Sin

embargo, cuando MySQL es utilizado por sistemas con altas cargas de consultas (SELECT's o INSERT's), es necesario modificar esta configuración para obtener un mejor desempeño. Existe un gran número de variables que se pueden configurar para alterar el funcionamiento de MySQL y probar todas las combinaciones es casi imposible. Para ello, se hizo una revisión de la literatura para tomar en cuenta los más importantes y los que podían tener un mayor impacto sobre el rendimiento del motor de MySQL.

El libro titulado *High Performance MySQL* por Schwartz, Zaitsev y Tkachenko (2012), tiene un amplio contenido acerca de cómo mejorar el desempeño de una base de datos MySQL. Recomendaciones importantes que se mencionan en este documento incluyen optimizaciones a nivel de esquema de la BD, definición de índices sobre las columnas para optimizar las búsquedas, optimizaciones en la sintaxis de la consulta (a nivel de *query*), a nivel de operaciones I/O, uso de la memoria RAM, uso de la memoria caché, entre otros.

Necesariamente, se incrementó a 8000 el número máximo de conexiones que debe aceptar MySQL para soportar la alta de carga de consultas (*max_connections*). Este parámetro se puede utilizar para permitir más conexiones o para limitarlas y evitar sobre cargar el sistema (Schwartz *et al*., 2012). Por otro lado, cuando las solicitudes de conexión fallaban más de 100 veces, MySQL bloqueaba el DNS de la Lambda y las siguientes solicitudes eran rechazadas. Se aumentó el valor del parámetro "*max_connect_errors*" a 4000 para evitar este problema.

La configuración más importante para optimizar el rendimiento de una instancia de MySQL es la cantidad de memoria asignada al *"pool"* de InnoDB

(*innodb_buffer_pool_size*). El *buffer* del *"pool"* es un espacio en memoria RAM donde InnoDB almacena los datos de las tablas conforme se acceden, lo que permite un acceso más rápido a la data de uso frecuente, acelerando así el procesamiento. En servidores con capacidad de memoria RAM en GB, se suele usar el 80 % de la memoria para uso dedicado del *"pool"* (Oracle Corporation, 2021).

La estimación del porcentaje de memoria que se debe asignar al *"pool"* de MySQL es distinta para cada ambiente y depende de muchas variables como, por ejemplo: la memoria que necesita el SO para funcionar, con qué otros procesos (otros programas) comparte MySQL la memoria RAM, cuánta memoria puede llegar a ocupar el resultado de una consulta, cuánta memoria está asignada para otros parámetros de MySQL, entre otros. En el caso de la instancia de EC2 que se utilizó para la BD de resultados, se utilizó el 87.5% de los 32 GB disponibles de memoria RAM como se mostró en la Tabla V, ya que la instancia está dedicada exclusivamente para el servidor de MySQL.

Una vez que se definió el tamaño del *"pool"*, fue necesario configurar también el parámetro "*buffer_pool_instances*". Luego de revisar la literatura, se encontró que existe mucha ambigüedad sobre el cálculo del valor óptimo para este parámetro, por lo que se recomienda realizar un *benchmark* para encontrar la mejor configuración. Para la consulta de los resultados mediante la Lambda, se determinó que un valor de 32 optimiza el rendimiento de la Base de Datos. Es importante tomar en cuenta que la relación entre los 3 parámetros de configuración del *"pool"* de InnoDB debe respetar la Ecuación 1, donde N es un entero positivo.

Ecuación 1. **Cálculo de los parámetros del Buffer de InnoDB**

$$pool_size = N * pool_chunk_size * pool_instances$$

Fuente: Oracle. (2021). *Configuring InnoDB Buffer Pool Size.*

Cabe destacar que la documentación de MySQL indica que para instancias con más de 1GB asignada al "*innodb_buffer_pool_size*", el valor por defecto de "*innodb_buffer_pool_instances*" es de 8 (Oracle Corporation, 2021). Investigando un poco más sobre este parámetro, se encontró que los autores del libro "*High Performance MySQL*" también sugieren utilizar 8 instancias, aunque dejan en claro que probablemente no han logrado entender del todo cómo funciona este parámetro y no han tenido experiencia utilizando otros valores (Schwartz *et al*., 2012).

Por último, los otros dos parámetros que se ajustaron para optimizar el rendimiento de MySQL son "*innodb_read_io_threads*" y "*thread_cache_size*". El primero optimiza las operaciones de lectura/escritura desde/hacia el disco duro, mientras el segundo optimiza la inicialización de las conexiones manteniendo en caché una cierta cantidad de conexiones para futuros usos. Durante las pruebas se determinó que, en general se puede configurar el número de núcleos físicos del servidor como valor para "*innodb_read_io_threads*". Para la instancia "*c5.4xlarge*" se configuró un valor de 8 ya que posee 15 vCPUs, lo que significa que posee 8 núcleos físicos. Para el parámetro "*thread_cache_size*", es necesario realizar pruebas con la aplicación ya que depende del caso de uso. Para la instancia "*c5.4xlarge*" con una carga de 2000 consultas concurrentes, se determinó que el valor óptimo de este parámetro es de 250. El Escenario 11 del Apéndice 9 introduce estas variables y amplía un poco más su función.

Los autores Schwartz *et al*., resaltan la importancia de otra configuración de InnoDB que optimiza el rendimiento de MySQL, la cual se identifica como "*innodb_log_file_size*". El archivo *log* sirve para registrar los cambios que ocurren en la Base de Datos antes de ser ejecutados, de modo que si ocurre un error durante la transacción y el servicio de MySQL se detiene por completo, la próxima vez que arranca podrá aplicar de nuevo los cambios leyéndolos desde el archivo *log* (Schartz *et al*.). Cuando hay una alta concurrencia de consultas de tipo escritura y el valor de este parámetro es muy pequeño, el archivo puede alcanzar rápidamente el tamaño especificado en *bytes*, por lo que MySQL tiene que realizar operaciones I/O para cerrar el archivo y crear uno nuevo. Este proceso puede degradar el rendimiento de MySQL. Sin embargo, en este proyecto no se tomó en cuenta este parámetro ya que aplica únicamente en operaciones de tipo escritura, mientras la optimización que se realizó es para operaciones de lectura.

6.8. Pruebas de carga

La prueba final para validar el rendimiento de la arquitectura consistió en una prueba de carga de 2000 solicitudes a cada 10 segundos durante 5 minutos hacia la Lambda de consulta de resultados y de forma paralela, se ejecutó una prueba que crea un registro de resultado a cada 10 segundos durante el mismo periodo de tiempo. Como se mencionó en la presentación de los resultados, no fue posible probar al mismo tiempo la Lambda para consultar la información pública debido a las limitaciones de la licencia de Artillery. Sin embargo, se recalca que las dos Lambdas pueden tener la misma configuración y el rendimiento de la Lambda de la información pública debe ser igual o incluso mejor que el de la Lambda de resultados, ya que va a recibir menos solicitudes y la cantidad de información que obtiene de la BD es menor.

De acuerdo con los resultados de Artillery mostrados en la Figura 46, la Lambda de consulta de resultados logró resolver correctamente 59,967 solicitudes y generó 33 errores de *timeout*, lo que resulta en un nivel de disponibilidad del ~99.95 % como lo muestra el gráfico de disponibilidad de la Figura 48. Este resultado se obtuvo durante el Escenario 14 del Apéndice 9 y al comprobar que satisface el objetivo principal del proyecto al ser mayor al 99.9 % que se propuso, se decidió tomar la configuración de dicho Escenario como resultado final.

No se lograron resolver 33 solicitudes de 60,000 debido a la capacidad de la instancia de la BD, pero también hay que reconocer que la optimización de MySQL aún se puede mejorar para soportar las 2000 solicitudes a cada 10 segundos durante más tiempo y garantizar un nivel de disponibilidad del 100%. Como se mencionó anteriormente, hay una gran cantidad de parámetros que se pueden configurar para optimizar MySQL y en este proyecto se enfocó únicamente en la cantidad de memoria para el *"pool"* de InnoDB, la cantidad de *threads* en caché para optimizar las conexiones y la cantidad de *threads* para las operaciones de lectura desde el disco duro.

La Lambda se tuvo que configurar con un *timeout* de 18 segundos y una concurrencia reservada de 3890, y aunque el gráfico de la Figura 47 indica que el 99% de las solicitudes se resolvieron en menos de 10 segundos, aún sigue siendo un tiempo de espera considerablemente alto y podría ser interpretado como sistema no disponible o sistema lento. Con una instancia de MySQL optimizada al 100 %, un *timeout* de 10 segundos y la concurrencia reservada en 2000 debería ser suficiente para soportar una carga de 2000 solicitudes a cada 10 segundos.

Otro aspecto para tomar en cuenta en los tiempos de latencia que reporta Artillery en el gráfico de la Figura 47, es que cada tarea en AWS Fargate tuvo que realizar 100 solicitudes al mismo tiempo. Artillery configura cada tarea de AWS Fargate con 1 vCPU y 1GB de memoria RAM, por lo que podría haber una demora en el proceso de orquestar las 100 solicitudes y procesar los resultados. En un escenario ideal, cada tarea debería realizar una sola solicitud, de modo que sería necesario ejecutar 2000 tareas concurrentes. Sin embargo, la licencia de Artillery para desarrollador también tiene la limitación de permitir un máximo de 25 tareas al mismo tiempo en AWS Fargate. Para ejecutar hasta 1000 tareas y ejecutar múltiples pruebas al mismo tiempo en múltiples regiones de AWS, es necesario adquirir la licencia pagada, que a la fecha de realización de este trabajo tenía un costo de $819 al mes (Artillery Software Inc., 2021).

Después de realizar las pruebas, se comprobó que la arquitectura propuesta obliga a escalar verticalmente ya que, si se quiere soportar un mayor nivel de concurrencia, se debe aumentar los recursos de la instancia de la Base de Datos. Para corroborar esto, se realizó la prueba de carga de la Lambda de resultados en una instancia "*c5.9xlarge*" para la BD, la cual ofrece 36 vCPUs y 72GB de RAM. Los resultados de esta prueba se muestran en el Escenario 15 del Apéndice 15 y como se puede observar, se logró resolver exitosamente el 100 % de las solicitudes. En efecto, con un servidor de BD más potente, se logra resolver una mayor cantidad de solicitudes concurrentes. Con esta instancia, el *timeout* de la Lambda se redujo a 10 segundos y en las métricas de la Lambda que se muestran en la figura C del apéndice 15, el tiempo máximo de ejecución fue de ~8.92 segundos mientras el tiempo promedio se mantuvo por debajo de los 4 segundos. Para Artillery, el tiempo máximo fue de ~9.9 segundos y el 99% de las

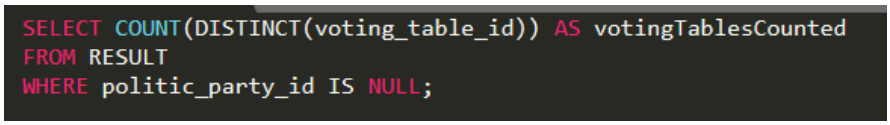
solicitudes registró un tiempo de latencia menor a ~6 segundos, tal como se aprecia en la figura B del apéndice 15.

No se mostraron los resultados del Escenario 15 del apéndice 9 como los resultados finales debido a que esa prueba no se ejecutó junto con la prueba de la API del sistema de tabulación, ya que fue imposible ejecutar todas las instancias de la arquitectura al mismo tiempo. La razón es que AWS impone un límite de 32 vCPUs para cada cuenta en cada región y aunque se logró una autorización para incrementar el límite a 36 vCPUs, únicamente dejó ejecutar la instancia "*c5.9xlarge*" a la vez. En resumen, es posible solicitar más unidades de procesador para una cuenta de AWS, pero la autorización puede tomar más tiempo y los costos se elevan sustancialmente.

6.9. Oportunidades de mejora

Tomando en cuenta los tipos de optimizaciones sugeridos por los autores del texto *High Performance MySQL*, se presenta un breve análisis de la base de datos del sistema de tabulación. La BD que se mostró en la Figura 18 tiene pocas tablas y se encuentra lo más normalizado posible, por lo que una optimización a nivel de esquema no aplica. La API para obtener los registros de resultados realiza consultas sencillas, filtros basados en IDs y en MySQL los IDs son automáticamente indexados, por lo que una optimización a nivel de índices no es necesaria. En la definición de las consultas, se trató de obtener únicamente la información que se necesita (seleccionando solo las columnas que importan), sin embargo, la consulta de los resultados aplica hasta 3 niveles de JOIN's para obtener la mesa, el centro de votación y el municipio, por lo que esta consulta podría ser un punto de mejora.

Figura 54. **Consulta para obtener las mesas contabilizadas**



```
SELECT COUNT(DISTINCT(voting_table_id)) AS votingTablesCounted
FROM RESULT
WHERE politic_party_id IS NULL;
```

Fuente: elaboración propia, utilizando Sublime Text.

Adicionalmente, la Figura 54 muestra la consulta para obtener la cantidad de mesas que ya fueron contabilizadas y como se puede ver, recorre todos los registros de resultados para verificar si la columna "*politic_party_id*" es nula. Definitivamente, esta consulta puede tener un impacto negativo en el tiempo de respuesta ya que los registros de resultados, aunque se ingresan totalizados, el número de registros podría alcanzar un valor de ~64,600 como lo muestra la Ecuación 2 o incluso más.

En el peor de los casos, todos los partidos políticos (19 según la Figura 38) tienen candidatura para todas las posiciones (el sistema de prototipo solo consideró a 3 pero en realidad son 5) en todos los municipios (340 en todo el país), y suponiendo que hay 2 centros de votación por cada municipio y una sola mesa por centro, el número aproximado de registros de resultados totalizados por mesa se puede calcular mediante la Ecuación 2.

Ecuación 2. **Cálculo aproximado del número de registros de resultados**

$$Registros = 5 * (2 * 340 * 19) = 64{,}600$$

Fuente: elaboración propia.

Por lo tanto, se podría reducir el tiempo de respuesta si se encuentra otra manera de guardar esta información para evitar esta consulta ineficiente. Una opción sería guardar en una tabla separada el valor de mesas contabilizadas y actualizarlo cada vez que se ingresa un registro de resultado donde el valor de la columna "*politic_party_id*" es nulo, de manera que la consulta de resultados solo tiene que localizar directamente este registro y leer el valor guardado.

En trabajos futuros se puede optimizar más la configuración de MySQL realizando pruebas con los distintos parámetros de configuración que dispone, ya que en este proyecto se experimentó principalmente con el rendimiento de InnoDB y el almacenamiento de *threads* en cache mediante la variable "*thread_cache_size*". Valdría la pena también sustituir MySQL por otro manejador de base de datos como PostgreSQL o una instancia en Amazon RDS para analizar su rendimiento comparado con MySQL. Amazon RDS se vende como un servicio de alto rendimiento, alta disponibilidad y seguridad (Amazon Web Services, Inc., 2021).

Finalmente, en el apéndice 10 se plantea una alternativa a la arquitectura que sustituye las Lambdas por Instancias de EC2 conectadas a un balanceador de cargas de tipo aplicación *Application load balancer* (*ALB*). Esta arquitectura, a diferencia de la que se propuso en la Figura 17, permite escalar horizontalmente ya que el grupo objetivo (*target*) del *ALB* permite registrar varias instancias. Un seguimiento a este trabajo podría ser configurar esta arquitectura y probarla con la misma carga que se planteó en los objetivos de este proyecto.

CONCLUSIONES

1. Se diseñó e implementó una arquitectura de alta disponibilidad en AWS que, complementado con una arquitectura lógica de microservicios, garantiza un nivel de disponibilidad mínimo del 99.9 % para la consulta de resultados durante una carga de 2000 usuarios concurrentes sin afectar la disponibilidad del sistema de tabulación.

2. La arquitectura de replicación *Master-Slave* ayudó a independizar los sistemas de tabulación y presentación de resultados al permitir la consulta de los resultados desde una BD secundaria y el ingreso de estos desde la BD primaria, mejorando así la disponibilidad de los datos.

3. Gracias a la implementación de las APIs siguiendo una arquitectura de microservicios, se independizó la consulta de los resultados del proceso de tabulación implementando dos aplicaciones web como *Single Page Application* (SPAs) desacopladas.

4. Mediante los servicios Lambda y EC2 de AWS, se implementó en la nube una arquitectura de alta disponibilidad, que complementado con una arquitectura de replicación Master-Slave en MySQL, garantiza un nivel de disponibilidad mínimo del 99.9 % durante la consulta de los registros de resultados.

5. Durante la prueba de carga, la arquitectura en AWS que soporta el microservicio para la consulta de resultados registró un nivel de disponibilidad del 99.95 %. Al mismo tiempo, la prueba de la API del sistema de tabulación obtuvo un nivel de disponibilidad del 100 % al procesar correctamente todas las solicitudes para crear registros de resultados.

RECOMENDACIONES

1. Luego de haber finalizado este proyecto, se descubrió que la arquitectura propuesta escala de forma vertical por lo que, para sistemas con una concurrencia de cientos de miles de solicitudes por segundo, no es recomendable. Para ello se recomienda probar la arquitectura que se propone en el Apéndice 15 que utiliza Balanceadores de Carga de tipo Aplicación (Application Load Balancer) para distribuir la carga de solicitudes entre varias instancias de EC2, permitiendo escalar de forma horizontal. Esta arquitectura también se basa en una configuración de réplica de Base de Datos de tipo Master-Slave pero con más nodos secundarios.

2. En futuros trabajos se recomienda experimentar con otros parámetros de configuración de MySQL para mejorar la optimización que se logró en este proyecto.

3. Otra manera de mejorar el rendimiento de la arquitectura propuesta sería levantar instancias de MySQL en Amazon RDS, que es un servicio administrado totalmente por AWS que garantiza un alto rendimiento y una alta disponibilidad.

REFERENCIAS

1. Álvarez, N. (27 de junio de 2019). Se registra falla en el sistema del TSE para la revisión de actas. *Publinews*. Recuperado de https://www.publinews.gt/gt/noticias/2019/06/27/falla-sistema-del-tse-la-revision-actas.html

2. Amazon Web Services, Inc. (2014). *Getting Started With AWS.* Washington: Autor. Recuperado de https://awsdocs.s3.amazonaws.com/gettingstarted/latest/awsgsg-intro.pdf

3. Amazon Web Services, Inc. (2019). *About AWS*. Whashington: Autor. Recuperado de https://aws.amazon.com/about-aws/

4. Amazon Web Services, Inc. (2019). *AWS Lambda*. AWS. Recuperado de https://aws.amazon.com/lambda/

5. Amazon Web Services, Inc. (2021). *Amazon Relational Database Service (RDS)*. AWS. Recuperado de https://aws.amazon.com/rds/

6. Amazon Web Services, Inc. (2021). *AWS Fargate*. AWS. Recuperado de https://aws.amazon.com/fargate

7. Amazon Web Services, Inc. (2021). *Lambda functions as targets*. AWS. Recuperado de https://docs.aws.amazon.com/elasticloadbalancing/latest/application/lambda-functions.html

8. Amazon Web Services, Inc. (2021). *Listeners for your Application Load Balancers*. AWS. Recuperado de https://docs.aws.amazon.com/elasticloadbalancing/latest/application/load-balancer-listeners.html

9. Amazon Web Services, Inc. (2021). *Optimize CPU options*. AWS. Recuperado de https://docs.aws.amazon.com/AWSEC2/latest/UserGuide/instance-optimize-cpu.html

10. Artillery Software Inc. (2021). *Artillery Pro*. Artillery. Recuperado de https://artillery.io/pro/

11. Artillery Software Inc. (2021). *Why Artillery?* Artillery. Recuperado de https://artillery.io/docs/guides/overview/why-artillery.html

12. Avram, M. G. (2014). Advantages and challenges of adopting cloud computing from an enterprise perspective. *Procedia Technology,* (12) 529-534. Recuperado de https://www.sciencedirect.com/science/article/pii/S221201731300710X

13. Baez, E. (2 de diciembre de 2020). *Function as a Service (FaaS) explained*. [Mensaje en un blog]. Recuperado de https://www.scalyr.com/blog/function-as-a-service-faas/

14. Cloudflare, Inc. (2019). *What Is Function as a Service (FaaS)?* Cloudflare. Recuperado de https://www.cloudflare.com/learning/serverless/glossary/function-as-a-service-faas/

15. De Paz, J. M. (2017). *Diseño e implementación de una Arquitectura escalable basada en Microservicios para un Sistema de gestión de Aprendizaje con características de red social.* (Tesis de maestría), Universidad San Carlos de Guatemala, Guatemala, Guatemala. Recuperado de http://www.repositorio.usac.edu.gt/7023/1/JOS%C3%89%20MANUEL%20DE%20PAZ%20ESTRADA.pdf

16. Evans, E. (2003). *Domain-Driven Design: Tackling Complexity in the Heart of Software.*. Oreilly Recuperado de https://www.oreilly.com/library/view/domain-driven-design-tackling/0321125215/

17. EVERAC99. (06 de abril de 2009). *Estimando usuarios concurrentes*. [Mensaje en un blog]. Recuperado de https://everac99.wordpress.com/2009/06/04/estimando-usuarios-concurrentes/

18. Fáctica. (24 de junio de 2019). Lo que sabemos hasta ahora de los problemas con el conteo de votos. *Nomada*. Recuperado de https://nomada.gt/pais/elecciones-2019/lo-que-sabemos-hasta-ahora-de-los-problemas-con-el-conteo-de-votos/

19. Fielding, T. (2000). *Architectural Styles and the Design of Network-based Software Architectures.* (Tesis doctoral), University of California, Irvine. California. Recuperado de https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf

20. Golden, B. (2013). *Amazon Web Services for Dummies.* Hoboken, New Yersey: John Wiley & Sons, Inc. Recuperado de https://fit.mta.edu.vn/files/DanhSach/Book_Amazon%20webservices%20for%20dummies.pdf

21. Hong, S., Srivastava, A., Shambrook, W., y Dumitras, T. (2018). *Go Serverless: Securing Cloud via Serverless Design Patterns*. USENIX Association. Recuperado de https://www.usenix.org/conference/hotcloud18/presentation/hong

22. Hurwitz, J., Kaufman, M., y Halper, F. (2012). *Cloud Services for Dummies.* Hoboken, New Yersey: John Wiley & Sons, Inc. Recuperado de https://www.ibm.com/cloud-computing/files/cloud-for-dummies.pdf

23. Huston, T. (2019). *What is Microservices?* [Mensaje en un blog]. Recuperado de https://smartbear.com/solutions/microservices/

24. Kappagantula, S. (2019). *What Is Microservices – Introduction To Microservice Architecture*. [Mensaje en un blog]. Recuperado de https://www.edureka.co/blog/what-is-microservices/

25. Microsoft. (2019). *¿Qué es la nube pública, privada e híbrida?* [Mensaje en un blog]. Recuperado de https://azure.microsoft.com/es-es/overview/what-are-private-public-hybrid-clouds/

26. Mishra, S. K., Sahoo, B., y Parida, P. P. (2 de febrero 2017). Load balancing in cloud computing: A big picture. *Journal of King Saud University – Computer and Information Sciences* 149-158. Recuperado de https://www.sciencedirect.com/science/article/pii/S1319157817303361

27. Morales, S. (25 de junio de 2019). El porqué variaron los datos entre las juntas receptoras de votos y el sistema digital del TSE. *PrensaLibre*. Recuperado de https://www.prensalibre.com/guatemala/politica/la-explicacion-del-por-que-cambiaron-los-datos-recabados-por-las-juntas-receptoras-de-votos-y-el-sistema-digital-del-tse/

28. Newman, S. (2015). *Building Microservices.* Sebastopol, California: O'Reilly Media, Inc. Recuperado de http://ce.sharif.edu/courses/96-

97/1/ce924-1/resources/root/Books/building-microservices-designing-fine-grained-systems.pdf

29. Oracle Corporation. (2021). *Buffer Pool*. [Mensaje en un blog]. Recuperado de https://dev.mysql.com/doc/refman/8.0/en/innodb-buffer-pool.html

30. Oracle Corporation. (mayo 2021). *InnoDB Startup Options and System Variables*. [Mensaje en un blog]. Recuperado de https://dev.mysql.com/doc/refman/8.0/en/innodb-parameters.html#sysvar_innodb_buffer_pool_instances

31. Oracle Corporation. (mayo 2021). *Server System Variables*. [Mensaje en un blog]. Recuperado de https://dev.mysql.com/doc/refman/8.0/en/server-system-variables.html

32. Poccia, D. (2017). *AWS Lambda In Action: Event-driven Serverless applications.* Shelter Island, New York: Manning Publications Co. Recuperado de https://www.manning.com/books/aws-lambda-in-action

33. Richardson, C. (enero 2019). *What are microservices?* [Mensaje en un blog]. Recuperado de https://microservices.io

34. Sajquin, M. (junio de 2019). *Anuncian revisión de actas de Juntas Receptoras de Votos*. [Mensaje en un blog]. Recuperado de https://www.canalantigua.tv/anuncian-revision-de-actas-de-juntas-receptoras-de-votos/

35. Schwartz, B., Zaitsev, P., y Tkachenko, V. (2012). *High Performance MySQL.* 1005 Gravenstein Highway North, Sebastopol, United States: O'Reilly Media, Inc. Recuperado de https://www.oreilly.com/library/view/high-performance-mysql/9781449332471/

36. Solano, M. (19 de julio de 2019). El pulso del TSE e informáticos por verificar que las fallas en conteo de actas no se repita. *Nómada*. Recuperado de https://nomada.gt/pais/elecciones-2019/el-pulso-del-tse-e-informaticos-por-verificar-que-las-fallas-en-conteo-de-actas-no-se-repita/

37. Wibowo, A., Diana, Subekti, M., y Hendro. (2018). Building Scalable and Resilience Database System to Mitigate Disaster and Performance Risks. *Procedia Computer Science,* (135) 25-34. Recuperado de https://www.sciencedirect.com/science/article/pii/S1877050918314327

38. Zaymus, M. (2017). *Decomposition of monolithic web application to microservices.* (Tesis de licenciatura), JAMK University of Applied

Sciences, Kajaani, Finlandia. Recuperado de https://www.theseus.fi/handle/10024/131110

APÉNDICES

Apéndice 1. **Configuración de instancias de EC2**

A continuación, se documenta el proceso de configuración de MySQL en las instancias de EC2. El prefijo “$” significa que la instrucción se ejecuta desde la línea de comandos, mientras que el prefijo “mysql >” indica que la instrucción se ejecuta en el cliente de MySQL. Para levantar las instancias EC2 de las bases de datos, se recomienda seguir la documentación de AWS. En esta sección se documenta únicamente el proceso para instalar y configurar el servidor de MySQL.

1. Instalar mysql-server.

```
$ sudo su
$ yum update -y
$ wget https://dev.mysql.com/get/mysql80-community-release-el7-
3.noarch.rpm
$ yum localinstall mysql80-community-release-el7-3.noarch.rpm -y
$ yum install mysql-community-server -y
$ systemctl start mysqld.service
$ rm mysql80-community-release-el7-3.noarch.rpm
```

2. Copiar contraseña temporal de usuario *root*.

```
$ sudo grep 'temporary password' /var/log/mysqld.log
```

Continuación apéndice 1.

3. Cambiar contraseña de usuario *root*. Correr el siguiente comando el cuál solicita la contraseña temporal y luego la nueva contraseña. A las siguientes preguntas del comando, se puede contestar con una “y” para indicar “Sí”.

```
$ mysql_secure_installation
```

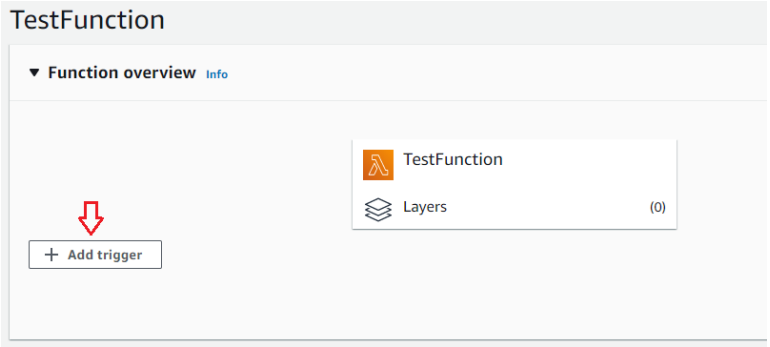
Fuente: elaboración propia.

Apéndice 2. **Configuración de un Trigger de tipo API Gateway para una Lambda**

API Gateway es otro servicio de AWS que permite crear APIs HTTP o REST. Para asociar una API a la función Lambda se puede hacer desde la página de *API Gateway* registrando la integración al momento de crear la API o desde la consola de AWS Lambda como se muestra en la figura A.

Continuación apéndice 2.

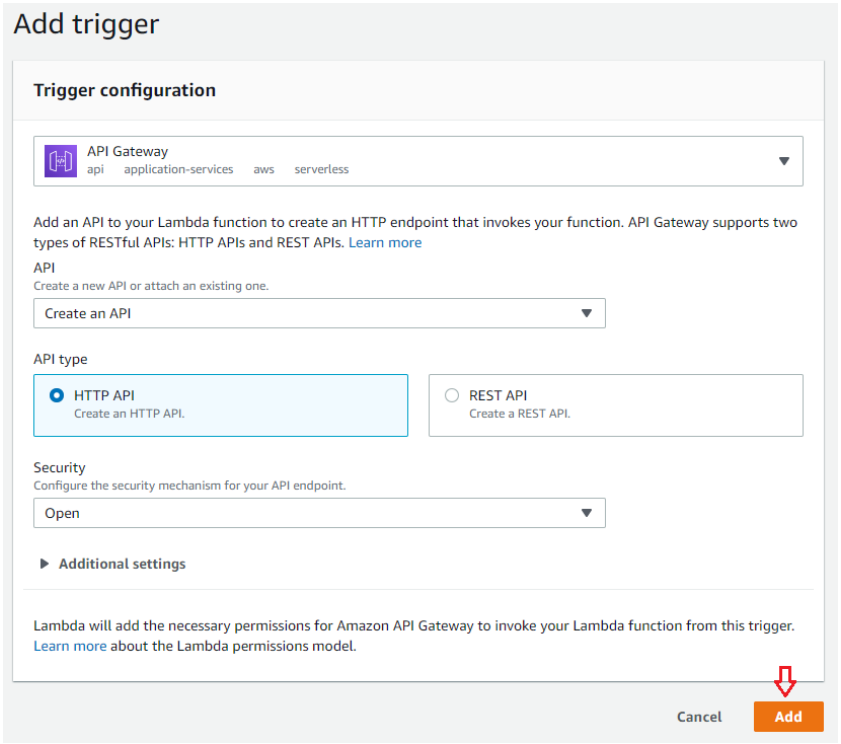
Figura A. **Registrar una API Gateway a una Lambda – Paso 1**



1. Desde la página de edición de la Lambda, hacer clic en "*Add trigger*".
2. En la pantalla para configurar el *trigger*, seleccionar *API Gateway* y aplicar las configuraciones que se muestran en la figura B. En la sección de "*Additional settings*" es necesario habilitar la configuración de "*CORS*". Luego hacer clic en "*Add*".
3. El resultado del paso anterior se muestra en la figura C. La URL para invocar la Lambda es la que aparece en *API endpoint*. Nótese que el método HTTP que registra por defecto para invocar la Lambda es *ANY*. Para configurar únicamente el método *GET* es necesario registrar la API desde la página de *API Gateway*.

Continuación apéndice 2.

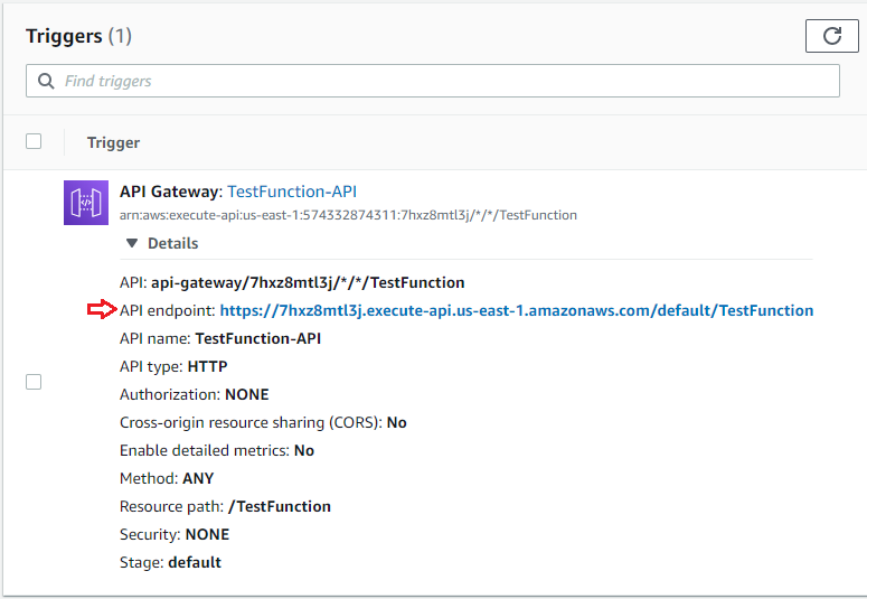
Figura B. **Registrar una API Gateway a una Lambda – Paso 2**



.

Continuación apéndice 2.

Figura C. **Registrar una API Gateway a una Lambda – Paso 2**



Fuente: Fuente: elaboración propia, utilizando consola de AWS Lambda.

Apéndice 3. **Configuración de una BD primaria**

1. Configurar la IP de las instancias EC2 de las BDs secundarias en el archivo "*/etc/hosts*". Esto va a permitir usar los nombres en vez de las IPs al momento de crear los usuarios de réplica en la BD primaria, de modo que cuando la IP de la instancia cambia, no es necesario actualizar en la BD, sino solo se actualiza el archivo "*/etc/hosts*". En este proyecto se usaron las IPs privadas, ya que estas nunca cambian. Sin embargo, cuando se tiene que usar las IPs públicas, esta forma de configuración resulta muy útil.

Continuación apéndice 3.

```
// /etc/hosts
...
172.31.58.153   public-data-db-host
3.89.47.240     results-db-host
172.31.60.96    votegt-api-host
```

2. Crear el archivo *log* que se va a especificar en la configuración de MySQL.

```
$ sudo su
$ mkdir /var/log/mysql
$ touch /var/log/mysql/mysql-bin.log
$ chown -R mysql:mysql /var/log/mysql
$ chmod 640 /var/log/mysql/mysql-bin.log
```

3. Crear el archivo de configuración de MySQL.

```
$ sudo su
$ mkdir /etc/mysql
$ nano /etc/mysql/my.cnf
```

4. Pegar en el archivo la configuración de la BD primaria.

```
[mysqld]
# Master DB configuration
server-id = 1
log-bin = /var/log/mysql/mysql-bin.log
tmpdir = /tmp
```

Continuación apéndice 3.

```
binlog_format = ROW
max_binlog_size = 500M
sync_binlog = 1
expire-logs-days = 7
slow_query_log
```

5. Reiniciar *mysqld*.

```
$ sudo systemctl restart mysqld
```

6. Acceder al cliente de MySQL y crear los usuarios de conexión de las BDs secundarias con permisos de replicación. Además, crear el usuario para la API del sistema de tabulación.

```
CREATE USER votegt_public_data_api@'public-data-db-host' IDENTIFIED
WITH caching_sha2_password BY 'SecretPassword';
GRANT REPLICATION SLAVE ON *.* TO votegt_public_data_api@'public-
data-db-host';
CREATE USER votegt_results_api@'results-db-host' IDENTIFIED WITH
caching_sha2_password BY 'SecretPassword';
GRANT REPLICATION SLAVE ON *.* TO votegt_results_api@'results-db-
host';
FLUSH PRIVILEGES;
CREATE USER votegt_api@'votegt-api-host' IDENTIFIED BY
'SecretPassword';
```

Continuación apéndice 3.

```
GRANT ALL PRIVILEGES ON votegt_2021.* TO votegt_api@'votegt-api-
host';
```

Fuente: elaboración propia.

Apéndice 4. **Configuración de una BD secundaria**

Como recomendación importante, se debe configurar primero la conexión de las BDs secundarias con la BD primaria antes de crear la BD del sistema de tabulación. De lo contrario, se tendrá que copiar manualmente la BD a las BDs secundarias para que el proceso de sincronización se aplique correctamente.

1. Al igual con la BD primaria, configurar la IP de la instancia de la BD primaria en el archivo "*/etc/hosts*".

```
// /etc/hosts
...
172.31.55.163   master-db-host
```

2. Crear el archivo *log* que se va a especificar en la configuración de MySQL.

```
$ sudo su
$ mkdir /var/log/mysql
$ touch /var/log/mysql/mysql-bin.log
$ chown -R mysql:mysql /var/log/mysql
$ chmod 640 /var/log/mysql/mysql-bin.log
```

Continuación apéndice 4.

3. Crear el archivo de configuración de MySQL.

```
$ sudo su
$ mkdir /etc/mysql
$ nano /etc/mysql/my.cnf
```

4. Pegar en el archivo la configuración de la BD secundaria. Esta configuración es la que se usó en la instancia de la BD de resultados durante las pruebas finales. Por cada BD, usar un "*server-id*" diferente.

```
[mysqld]
# Allow remote access
bind-address = 0.0.0.0

# Allow more connections (default is 151)
max_connections = 8100

# Allow more connection retries
max_connect_errors = 4000

# Configure pool: Use 87.5% of a 32GB memory (28GB)
innodb_buffer_pool_chunk_size = 939524096
innodb_buffer_pool_instances = 32
innodb_buffer_pool_size = 30064771072

# Increase thread cache size
thread_cache_size = 250

# Increase read io threads (default is 4)
innodb_read_io_threads = 8
```

Continuación apéndice 4.

```
# Slave DB configuration
server-id = 2
read_only = 1
log_bin = /var/log/mysql/mysql-bin.log
tmpdir = /tmp
binlog_format = ROW
max_binlog_size = 500M
sync_binlog = 1
expire-logs-days = 7
slow_query_log = 1
```

5. Reiniciar *mysqld*.

```
$ sudo systemctl restart mysqld
```

6. Desde la BD primaria correr el *query* "*SHOW MASTER STATUS\G;*" y usando la salida de este comando, configurar la BD secundaria para que pueda conectarse a la BD primaria. Usar el usuario que corresponde.

```
CHANGE MASTER TO MASTER_HOST='master-db-host',
MASTER_USER='votegt_results_api', MASTER_PASSWORD='SecretPassword',
MASTER_LOG_FILE='mysql-bin.000001', MASTER_LOG_POS=156;
```

Continuación apéndice 4.

7. Iniciar el proceso de réplica en la BD secundaria. Antes de hacer esto, es necesario autenticarse a la BD primaria manualmente, de lo contrario, el proceso de replicación no logra conectarse. La razón de esto es que los usuarios que se crearon en la BD primaria, se registraron con el método de autenticación "*caching_sha2_password*".

```
START REPLICA;
```

8. Verificar que el proceso de réplica está corriendo y la BD secundaria está conectada a la BD primaria. Al correr el siguiente query, los valores de las variables "*Replica_IO_Running*" y "*Replica_SQL_Running*" deberían decir "*Yes*".

```
SHOW REPLICA STATUS\G;
```

9. Crear en la BD secundaria el usuario que va a utilizar la Lambda para conectarse. Las instrucciones siguientes corresponden a la Lambda de consulta de resultados.

```
CREATE USER votegt_results_api@'%' IDENTIFIED WITH
mysql_native_password BY 'SecretPassword';
GRANT SELECT ON votegt_2021.* TO 'votegt_results_api'@'%';
FLUSH PRIVILEGES;
```

Fuente: elaboración propia.

Apéndice 5. **Configuración del servidor de Nginx para la API del sistema de tabulación**

La mayoría de las configuraciones son las que ya trae *nginx* por defecto. Las más importantes que se agregaron fueron:

- Ejecutar *nginx* con el usuario “ec2-user” para evitar errores de permisos.
- La primera configuración de “*server*” corresponde al puerto 80 (HTTP) y redirecciona todas las solicitudes al dominio con HTTPS.
- La segunda configuración de “*server*” corresponde al puerto 443 (HTTPS) y es donde se especifica el nombre del dominio y se configura el certificado de seguridad.

```
user ec2-user;
worker_processes auto;
error_log /var/log/nginx/error.log;
pid /run/nginx.pid;

# Load dynamic modules. See /usr/share/doc/nginx/README.dynamic.
include /usr/share/nginx/modules/*.conf;

events {
    worker_connections 1024;
}

http {
    log_format  main  '$remote_addr - $remote_user [$time_local] "$request" '
                      '$status $body_bytes_sent "$http_referer" '
                      '"$http_user_agent" "$http_x_forwarded_for"';

    access_log  /var/log/nginx/access.log  main;

    sendfile            on;
```

Continuación apéndice 5.

```
    tcp_nopush          on;
    tcp_nodelay         on;
    keepalive_timeout   65;
    types_hash_max_size 4096;

    include             /etc/nginx/mime.types;
    default_type        application/octet-stream;

    # Load modular configuration files from the /etc/nginx/conf.d directory.
    # See http://nginx.org/en/docs/ngx_core_module.html#include
    # for more information.
    include /etc/nginx/conf.d/*.conf;

    server {
        listen         80 default_server;
        listen         [::]:80 default_server;
        server_name    votegt-api.servirsa.com www.votegt-api.servirsa.com;
        location / {
             return 301 https://$server_name$request_uri;
        }
    }

    # Settings for a TLS enabled server.

    server {
        listen       443 ssl;
        listen       [::]:443 ssl;
        server_name  votegt-api.servirsa.com www.votegt-api.servirsa.com;

        ssl_certificate "/etc/letsencrypt/live/votegt-
api.servirsa.com/fullchain.pem";
        ssl_certificate_key "/etc/letsencrypt/live/votegt-
api.servirsa.com/privkey.pem";

        ssl_session_cache shared:SSL:1m;
        ssl_session_timeout  10m;
        ssl_protocols TLSv1 TLSv1.1 TLSv1.2;
        ssl_ciphers 'EECDH+AESGCM:EDH+AESGCM:AES256+EECDH:AES256+EDH';
        ssl_prefer_server_ciphers on;
```

Continuación apéndice 5.

```
        # Load configuration files for the default server block.
        include /etc/nginx/default.d/*.conf;
        location / {
            proxy_pass http://127.0.0.1:3000;
            proxy_http_version 1.1;
            proxy_set_header Upgrade $http_upgrade;
            proxy_set_header Connection 'upgrade';
            proxy_set_header Host $host;
            proxy_cache_bypass $http_upgrade;
        }
    }
}
```

Fuente: elaboración propia.

Apéndice 6. **Configuración de Artillery para la prueba de carga de la Lambda de resultados**

Esta configuración genera 2,000 solicitudes entre 20 tareas de AWS Fargate (100 cada una) a cada 10 segundos y lo hace 30 veces para simular 5 minutos de carga. En total genera 60,000 solicitudes.

```
config:
  target: 'https://lmjaptb3tc.execute-api.us-east-1.amazonaws.com'
  http:
    timeout: 60

  phases:
    - duration: 1
      arrivalCount: 100
      name: 'Get results 1'
```

Continuación apéndice 6.

```
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 2'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 3'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 4'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 5'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 6'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 7'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 8'
    - pause: 10
    - duration: 1
      arrivalCount: 100
```

Continuación apéndice 6.

```
      name: 'Get results 9'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 10'
    - duration: 1
      arrivalCount: 100
      name: 'Get results 11'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 12'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 13'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 14'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 15'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 16'
    - pause: 10
    - duration: 1
      arrivalCount: 100
```

Continuación apéndice 6.

```
      name: 'Get results 17'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 18'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 19'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 20'
    - duration: 1
      arrivalCount: 100
      name: 'Get results 21'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 22'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 23'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 24'
    - pause: 10
    - duration: 1
      arrivalCount: 100
```

Continuación apéndice 6.

```
      name: 'Get results 25'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 26'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 27'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 28'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 29'
    - pause: 10
    - duration: 1
      arrivalCount: 100
      name: 'Get results 30'

scenarios:
  - name: 'Get vote results'
    flow:
      - get:
          url: '/v1/results?positionId=1'
```

Fuente: elaboración propia.

Apéndice 7. **Configuración de Artillery para la prueba de la API del sistema de tabulación**

Esta configuración manda a crear un registro de resultado a la API del sistema de tabulación a cada 10 segundos y lo hace 30 veces para realizar una prueba de 5 minutos. La data que manda en las solicitudes la obtiene del archivo *results.csv*, lee línea por línea y carga el valor de cada columna en variables. Además, las solicitudes necesitan ir autenticadas por lo que es necesario generar un *token* de sesión y exportarlo desde la línea de comandos con el nombre API_TOKEN.

```
config:
  target: 'https://votegt-api.servirsa.com'
  https:
    rejectUnauthorized: false
  http:
    timeout: 60
    pool: 1
  payload:
    path: 'results.csv'
    fields:
      - voting_table_id
      - position_id
      - politic_party_id
      - validVotes
      - nullVotes
      - blankVotes
    order: 'sequence'
    skipHeader: true
  defaults:
    headers:
```

Continuación apéndice 7.

```
      authorization: 'Bearer {{ $processEnvironment.API_TOKEN }}'

  # 5 minutes, 1 request each 10 seconds => 30 requests
  phases:
    - name: 'Post result record 1'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 2'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 3'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 4'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 5'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 6'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 7'
      duration: 1
      arrivalCount: 1
```

Continuación apéndice 7.

```
    - pause: 10
    - name: 'Post result record 8'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 9'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 10'
      duration: 1
      arrivalCount: 1
    - name: 'Post result record 11'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 12'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 13'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 14'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 15'
      duration: 1
```

Continuación apéndice 7.

```
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 16'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 17'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 18'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 19'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 20'
      duration: 1
      arrivalCount: 1
    - name: 'Post result record 21'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 22'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 23'
      duration: 1
```

Continuación apéndice 7.

```
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 24'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 25'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 26'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 27'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 28'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 29'
      duration: 1
      arrivalCount: 1
    - pause: 10
    - name: 'Post result record 30'
      duration: 1
      arrivalCount: 1
  processor: './artillery-processor.js'
scenarios:
```

Continuación apéndice 7.

```
  - name: 'Post result record'
    flow:
      - log: 'POST data: { voting_table_id: {{ voting_table_id }},
position_id: {{ position_id }}, politic_party_id: {{
politic_party_id }}, validVotes: {{ validVotes }} }'
      - post:
          url: '/results'
          json:
            voting_table_id: '{{ voting_table_id }}'
            position_id: '{{ position_id }}'
            politic_party_id: '{{ politic_party_id }}'
            validVotes: '{{ validVotes }}'
            nullVotes: '{{ nullVotes }}'
            blankVotes: '{{ blankVotes }}'
          afterResponse: 'printStatus'
```

Fuente: elaboración propia.

Apéndice 8. **Pruebas de Artillery con AWS Fargate**

Prerrequisitos:

- Instalar las librerías de NodeJS "*artillery@latest*" y "*artillery-pro@latest*".
- Obtener una licencia de desarrollador desde la página de Artillery.io.
- Configurar el CLI de AWS en la computadora desde donde se van a ejecutar las pruebas.
- Crear desde la consola de IAM un usuario que pertenezca a un grupo con el permiso *AdministratorAccess*.

Continuación apéndice 8.

Si ya se completaron los pasos anteriores, ya se puede correr el comando de Artillery que configura todos los recursos en AWS que el *cluster* necesita para ejecutar las pruebas. Nótese que es necesario contar con la licencia de Artillery para ejecutar este comando.

```
$ artillery deploy --region us-east-1 --license-key-file
~/Downloads/artillery-pro-cluis.usac_at_gmail.com.license_key.txt
```

Luego es necesario registrar al usuario creado en AIM el permiso que Artillery necesita para ejecutar las tareas en AWS Fargate. Suponiendo que el nombre del usuario es “cluis”, el comando a ejecutar sería el siguiente.

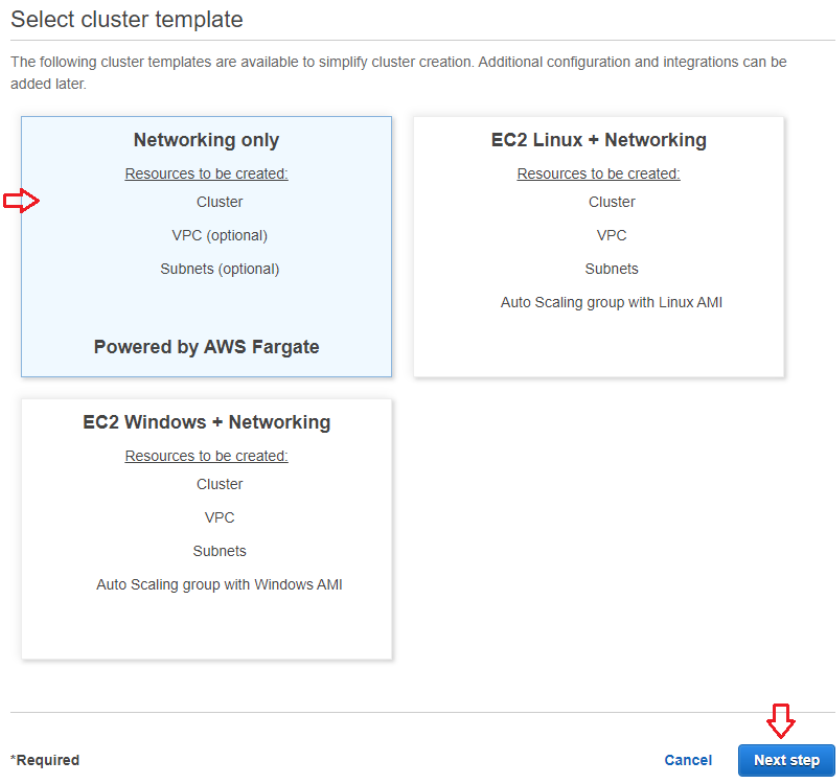
```
aws iam attach-user-policy --user-name cluis --policy-arn
arn:aws:iam::574332874311:policy/artilleryio-cli-user
```

A continuación, para poder ejecutar las pruebas en AWS Fargate, es necesario crear el *cluster* manualmente desde la consola del servicio *Elastic Container Service o ECS* y consiste básicamente en dos pasos.

1. Desde la consola de ECS, hacer clic en “*Create cluster*”. En la primera pantalla, seleccionar la primera opción “*Network only*” para utilizar contenedores de AWS Fargate como se muestra en la figura A.

Continuación apéndice 8.

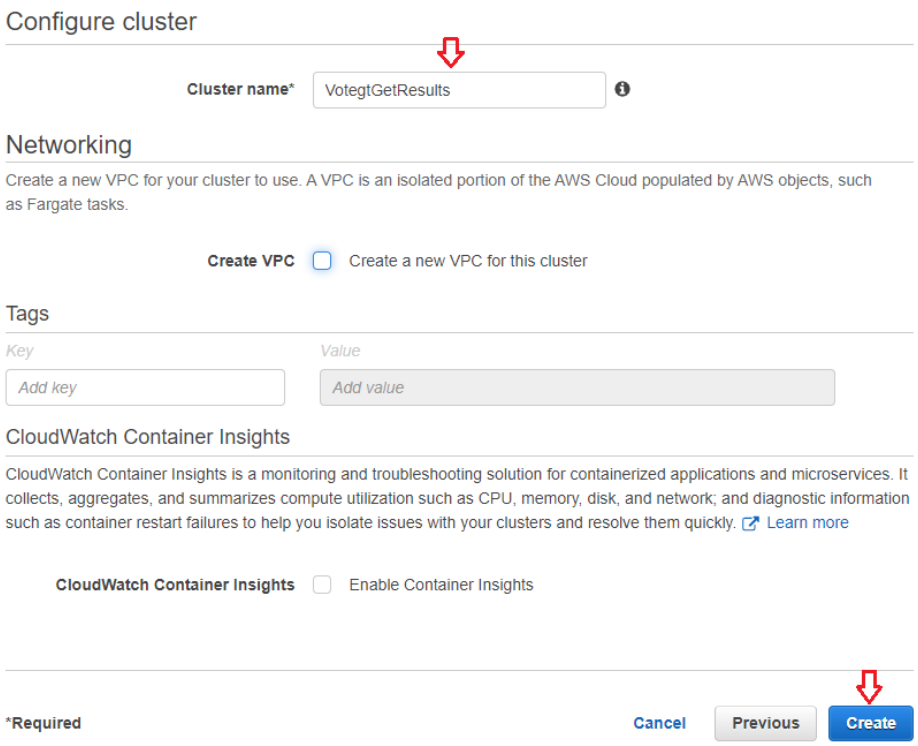
Figura A. **Crear clúster de ECS – Paso 1**



2. En el segundo paso, solo es necesario especificar el nombre del *cluster* como se observa en la Figura B y hacer clic en "*Create*".

Continuación apéndice 8.

Figura B. **Crear clúster de ECS – Paso 2**



El *cluster* ya está listo para que Artillery lo use para ejecutar las pruebas. Se usa el siguiente comando para ejecutar las pruebas contra la Lambda de resultados utilizando el *cluster* creado anteriormente y especificando 20 tareas.

Continuación apéndice 8.

```
artillery run-test test-results-api.yaml --cluster VotegtGetResults
--region us-east-1 --subnet-ids subnet-78b18835,subnet-
9c9df7c3,subnet-29fa4618 --launch-type ecs:fargate --count 20 --
output test-results-api.json
```

Luego se usa el siguiente comando para generar el reporte en formato HTML a partir del JSON generado por Artillery.

```
artillery report test-results-api.json --output test-results-
api.html
```

Para más información acerca de Artillery Pro y las opciones que se pasan al comando para correr la prueba en AWS Fargate, se recomienda consultar la documentación en Aritllery.io.

Fuente: elaboración propia, utilizando consola de ECS.

Apéndice 9. **Pruebas realizadas para encontrar la configuración óptima de la arquitectura**

Escenario 1

La siguiente tabla muestra la configuración que se utilizó para realizar la primera prueba desde una computadora personal. Esta prueba tuvo por objetivo encontrar cuántas solicitudes puede procesar una computadora. Tanto la Lambda como MySQL se dejó con los valores por defecto.

Continuación apéndice 9.

Configuración de escenario 1

Descripción	500 solicitudes desde laptop personal.
Instancia EC2	
Tipo	t2.micro
vCPUs	1
Memoria RAM	8GB
MySQL JS	
connectionLimit	1000
MySQL	
max_connections	151 (default)
Lambda	
Timeout	3s
Reserved concurreny	1000
Artillery	
Timeout*	120s

Fuente: elaboración propia.

La figura A muestra los resultados generados por Artillery y como se puede observar, únicamente se lograron resolver exitosamente 219 solicitudes mientras 281 terminaron por *timeout* de Artillery.

Continuación apéndice 9.

Figura A. **Resultados de Artillery**

Summary

Test duration	150 sec
Scenarios created	500
Scenarios completed	219

Scenario counts:

Request for results	500 (100%)

Codes

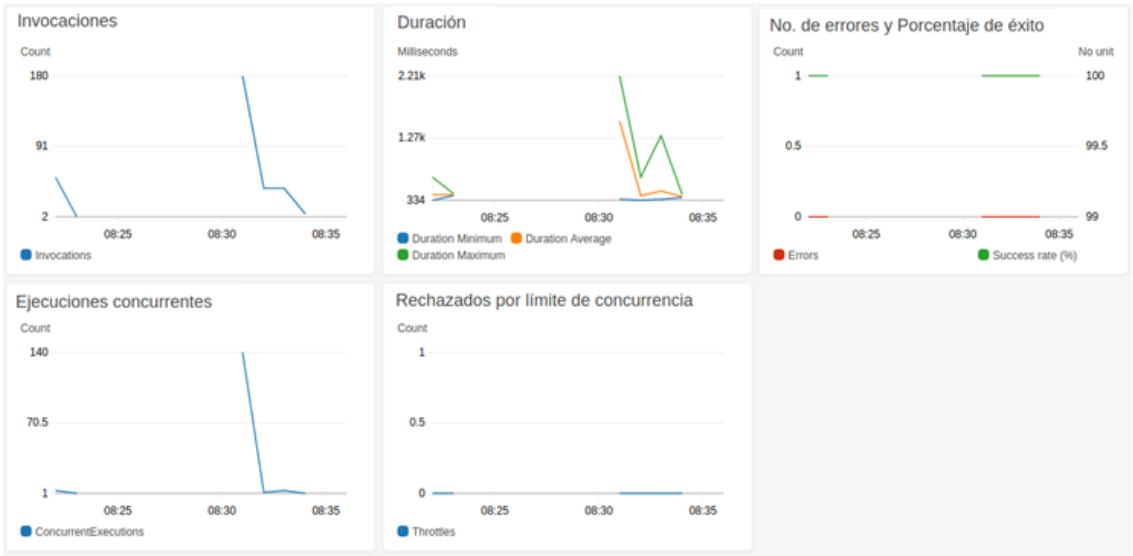
200	219

Errors

ETIMEDOUT	281

Fuente: elaboración propia, usando comando de Artillery para generar reportes.

Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Figura C. **Monitoreo de EC2**

Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Luego de varias iteraciones, se comprobó que una computadora no podía resolver 500 solicitudes con estas configuraciones. Los errores de tipo ETIMEDOUT reportados por Artillery indican que 281 solicitudes no se lograron resolver en 120s. De acuerdo con las métricas de la Lambda, no hubo ningún error, lo que indica que las 500 solicitudes si se lograron resolver correctamente, pero muchos tardaron más de 120s.

El gráfico de errores en la figura B incluye errores en tiempo de ejecución del código, errores durante las consultas a la BD o cuando la ejecución de la función no finaliza en el tiempo configurado para la Lambda (*timeout*). Por otro lado, en la figura C se observa que el porcentaje de uso del procesador de la instancia de EC2 fue mínimo. En los escenarios siguientes, los parámetros con un asterisco (*) indican que cambiaron respecto al escenario anterior.

Continuación apéndice 9.

Escenario 2

En el escenario descrito por la siguiente tabla se usó la misma configuración del Escenario 1 pero solo se programaron 200 solicitudes desde una computadora personal.

Configuración de escenario 2

Descripción*	200 solicitudes desde laptop personal.
Instancia EC2	
Tipo	t2.micro
vCPUs	1
Memoria RAM	8GB
MySQL JS	
connectionLimit	1000
MySQL	
max_connections*	1000
Lambda	
Timeout	3s
Reserved concurreny	1000
Artillery	
Timeout*	60s

Fuente: elaboración propia.

Continuación apéndice 9.

Figura A. **Resultados de Artillery**

Summary

Test duration	130 sec
Scenarios created	200
Scenarios completed	200

Scenario counts:

Request for results	200 (100%)

Codes

200	200

Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

Figura B. **Métricas de la Lambda**

Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Figura C. **Monitoreo de EC2**

Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Luego de varias pruebas con esta configuración, se logró verificar que 200 solicitudes pueden ser correctamente procesadas por una computadora como se muestra en la figura A. Además, se configuró MySQL para que permitiera hasta 1000 conexiones al mismo tiempo ya que en la bitácora de la Lambda se podía visualizar el error "*Host '172.31.58.153' is blocked because of many connection errors*". Las métricas de la Lambda en la figura B no muestran ninguna anomalía al igual que las métricas de la instancia de EC2 en la figura C.

Escenario 3

Una vez se logró verificar el número máximo de solicitudes que puede procesar exitosamente una sola computadora, se procedió a realizar una prueba con AWS Fargate utilizando la misma configuración y creando una sola tarea. La configuración completa de la prueba se muestra a continuación.

Continuación apéndice 9.

Configuración de escenario 3

Descripción*	Una tarea de 200 solicitudes en AWS Fargate.
Instancia EC2	
Tipo	t2.micro
vCPUs	1
Memoria RAM	8GB
MySQL JS	
connectionLimit	1000
MySQL	
max_connections	1000
Lambda	
Timeout	3s
Reserved concurreny	1000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Continuación apéndice 9.

Figura A. **Resultados de Artillery**

Summary

Test duration	10 sec
Scenarios created	200
Scenarios completed	200

Scenario counts:

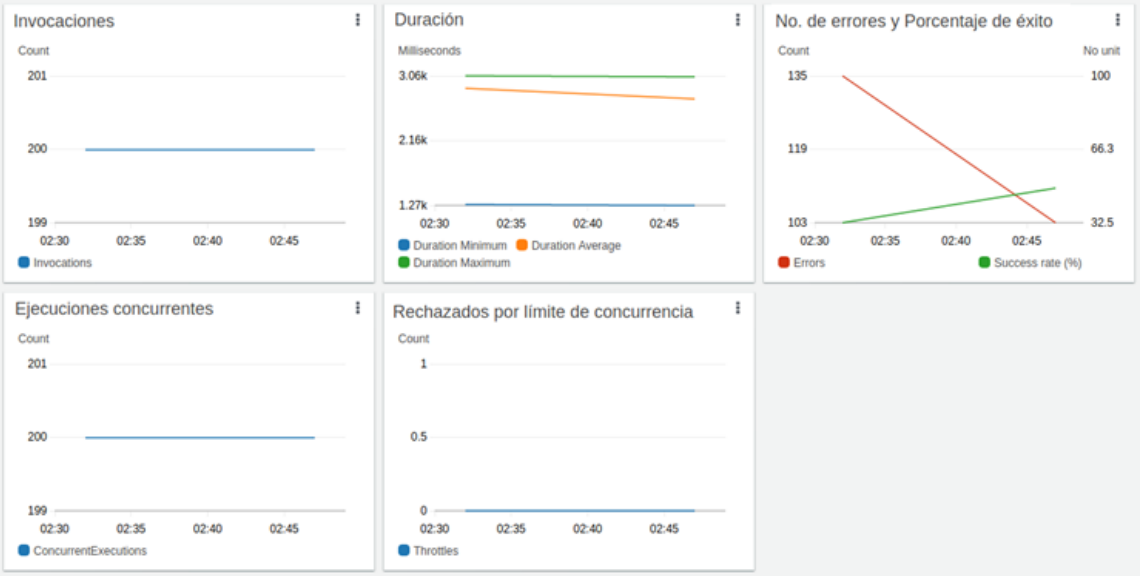
Request for results	200 (100%)

Codes

200	97
500	103

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

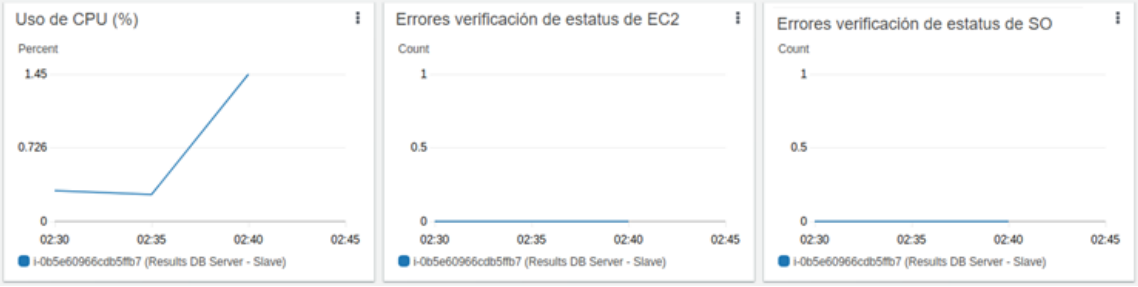
Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Figura C. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Como se puede observar en los resultados de Artillery en la figura A, una tarea en AWS Fargate pudo procesar únicamente 97 solicitudes correctamente y el resto falló por *timeout*. La Figura B muestra una gran cantidad de errores en la Lambda mientras que las métricas de EC2 en la figura C no muestran ninguna sobre carga del procesador.

Escenario 4

Dado que una tarea de AWS no lograba procesar 200 solicitudes, se decidió enviar únicamente 100 solicitudes a cada tarea como se muestra en la tabla a continuación, pero en esta ocasión se generaron 1000 solicitudes divididos entre 10 tareas.

Continuación apéndice 9.

Configuración de escenario 4

Descripción*	1000 solicitudes entre 10 tareas en AWS Fargate.
Instancia EC2	
Tipo	t2.micro
vCPUs	1
Memoria RAM	8GB
MySQL JS	
connectionLimit	1000
MySQL	
max_connections	1000
Lambda	
Timeout	3s
Reserved concurreny	1000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Continuación apéndice 9.

Figura A. **Resultados de Artillery**

Summary

Test duration	10 sec
Scenarios created	1000
Scenarios completed	1000

Scenario counts:

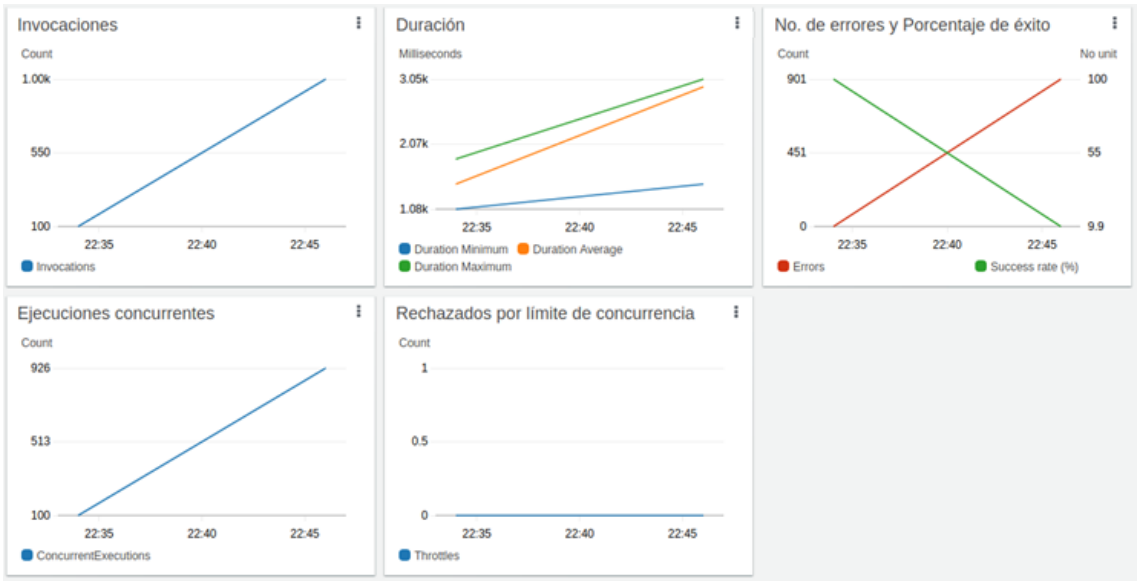
Request for results	1000 (100%)

Codes

200	99
500	901

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

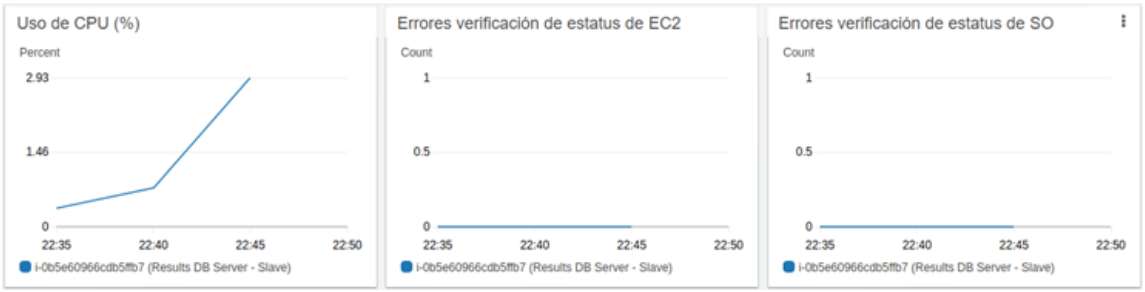
Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Figura C. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

La figura A muestra el reporte generado por Artillery e indica que se resolvieron únicamente 99 solicitudes y 901 resultaron en error. Por otro lado, las métricas de la Lambda en la Figura B indican que se recibieron las 1000 solicitudes y el gráfico de errores refleja las 901 solicitudes que fallaron. El uso del procesador en la figura C sigue estando bajo.

Lo que sucedió fue que la instancia de la BD ya sea, se tardó más de 3 segundos por cada solicitud o simplemente rechazó las conexiones. La Lambda, al tener un *timeout* de 3 segundos terminó reportándolos como errores. En la bitácora de Amazon CloudWatch Insights se identificó el siguiente error: "*connect ECONNREFUSED 172.31.25.132:3306*", lo cual indica que MySQL rechazó varios intentos de conexión.

Continuación apéndice 9.

Escenario 5

Para lograr resolver 1000 solicitudes fue necesario migrar la instancia de EC2 a una de tipo "*t2.xlarge*" que provee 4 unidades de CPU y 16GB de RAM. Para que MySQL pudiera aprovechar la RAM disponible, se configuró el *pool* de InnoDB para utilizar un 75 % (12GB) del total y creara 12 instancias, tal como se detalla en la siguiente tabla.

Configuración de escenario 5

Descripción	1000 solicitudes entre 10 tareas en AWS Fargate.
Instancia EC2*	
Tipo	t2.xlarge
vCPUs	4
Memoria RAM	16GB
MySQL JS	
connectionLimit	1000
MySQL	
max_connections	1000
max_connect_errors*	1000
innodb_buffer_pool_size*	12GB => 12884901888 B
innodb_buffer_pool_chunk_size*	1GB => 1073741824 B
innodb_buffer_pool_instances*	12
Lambda	
Timeout*	5s

Continuación apéndice 9.

Reserved concurreny	1000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Por otro lado, también se aumentó el *timeout* de la Lambda a 5 segundos ya que las consultas a la BD se estaban tardando más. También se configuró la variable "*max_connect_errors*" de MySQL para evitar que la Lambda termine siendo bloqueada por MySQL luego de varios intentos de conexión fallidos.

Figura A. **Resultados de Artillery**

Summary

Test duration	10 sec
Scenarios created	1000
Scenarios completed	1000

Scenario counts:

Request for results	1000 (100%)

Codes

200	1000

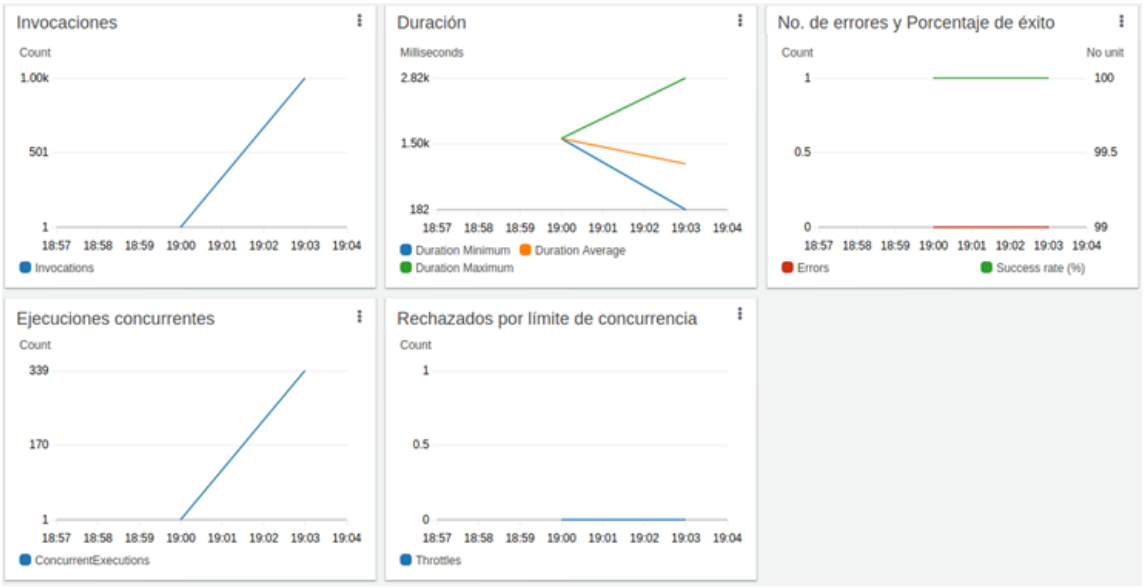
Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

Continuación apéndice 9.

Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Figura C. **Monitoreo de EC2**

Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Gracias a la personalización del *pool* de InnoDB se lograron resolver las 1000 solicitudes como muestran los resultados de Artillery en la figura A. Los gráficos en la figura B indican que la duración promedio de las ejecuciones en la Lambda fue menor a 1.5 segundos y la duración máxima fue de 2.8 segundos. Por su parte, la instancia de EC2 de la base de datos continuó reportando un uso mínimo del procesador de acuerdo con la figura C.

Escenario 6

Con la misma configuración del Escenario 5, se realizó la prueba final de 2000 solicitudes concurrentes entre 20 tareas en AWS Fargate (100 solicitudes por cada tarea) como se muestra a continuación.

Configuración de escenario 6

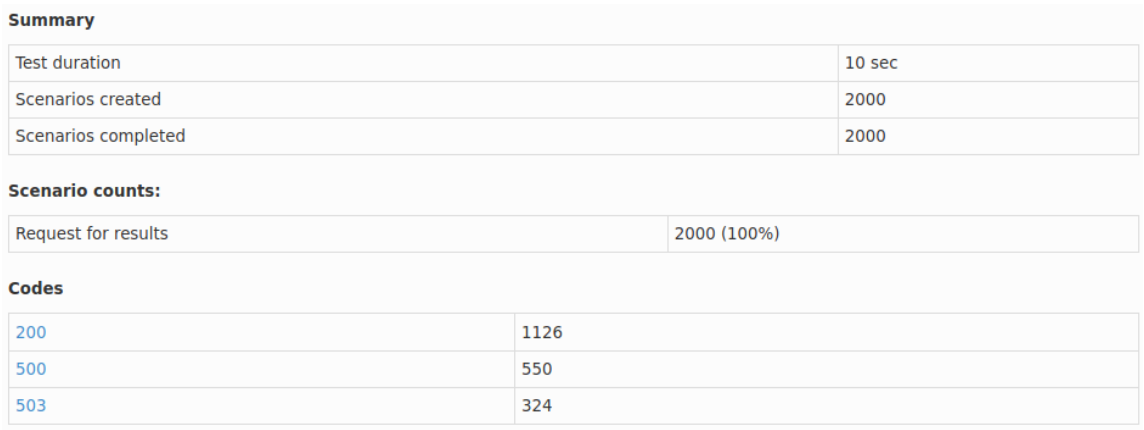
Descripción*	2000 solicitudes entre 20 tareas en AWS Fargate.
Instancia EC2	
Tipo	t2.xlarge
vCPUs	4
Memoria RAM	16GB
MySQL JS	
connectionLimit	1000
MySQL	
max_connections	1000
max_connect_errors	1000

Continuación apéndice 9.

innodb_buffer_pool_size	12GB => 12884901888 B
innodb_buffer_pool_chunk_size	1GB => 1073741824 B
innodb_buffer_pool_instances	12
Lambda	
Timeout	5s
Reserved concurreny	1000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Figura A. **Resultados de Artillery**



Summary

Test duration	10 sec
Scenarios created	2000
Scenarios completed	2000

Scenario counts:

Request for results	2000 (100%)

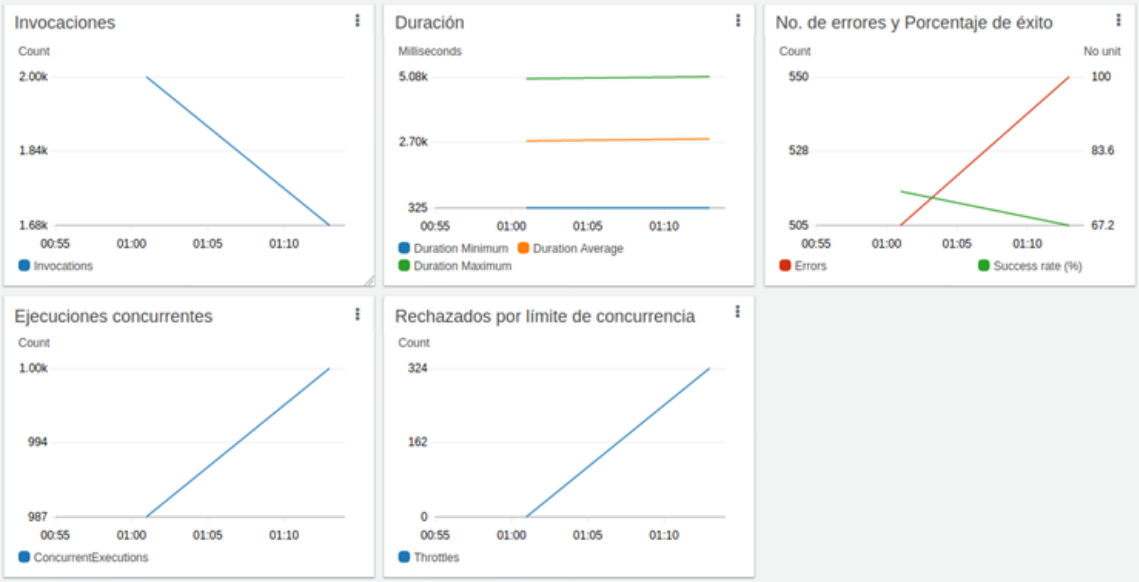
Codes

200	1126
500	550
503	324

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

Continuación apéndice 9.

Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Figura C. **Monitoreo de EC2**

Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Como lo muestran los resultados de Artillery en la figura A, no se resolvieron 550 solicitudes y se puede confirmar con las métricas de errores de la Lambda en la figura B. Además, se empezaron a generar errores de código 503 y en el gráfico "*Rechazados por límite de concurrencia*" se puede visualizar que estos errores se deben a que la Lambda alcanzó su límite de concurrencia (1000). Por otro lado, en la bitácora de eventos de la Lambda se visualizó de nuevo el error de MySQL "*ER_CON_COUNT_ERROR: Too many connections*". Por último, La figura C muestra que hubo un aumento en el porcentaje de uso del procesador, pero sigue siendo insignificante.

Escenario 7

En esta configuración se migró a una instancia de tipo "*t2.2xlarge*" para obtener hasta 32GB de RAM y poder utilizar 27GB para el *pool* de InnoDB, tal como se muestra en la siguiente tabla. Además, se aumentó el límite de conexiones de *mysqljs* para permitir hasta 2000 conexiones simultáneas.

Configuración de escenario 7

Descripción	2000 solicitudes entre 20 tareas en AWS Fargate.
Instancia EC2*	
Tipo	t2.2xlarge
vCPUs	8
Memoria RAM	32GB
MySQL JS	
connectionLimit*	2000
MySQL	

Continuación apéndice 9.

max_connections*	2000
max_connect_errors	1000
innodb_buffer_pool_size*	27GB => 28991029248 B
innodb_buffer_pool_chunk_size*	4.5GB => 4831838208 B
innodb_buffer_pool_instances*	6
Lambda	
Timeout	5s
Reserved concurreny	1000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Figura A. **Resultados de Artillery**

Summary

Test duration	10 sec
Scenarios created	2000
Scenarios completed	2000

Scenario counts:

Request for results	2000 (100%)

Codes

200	1867
500	133

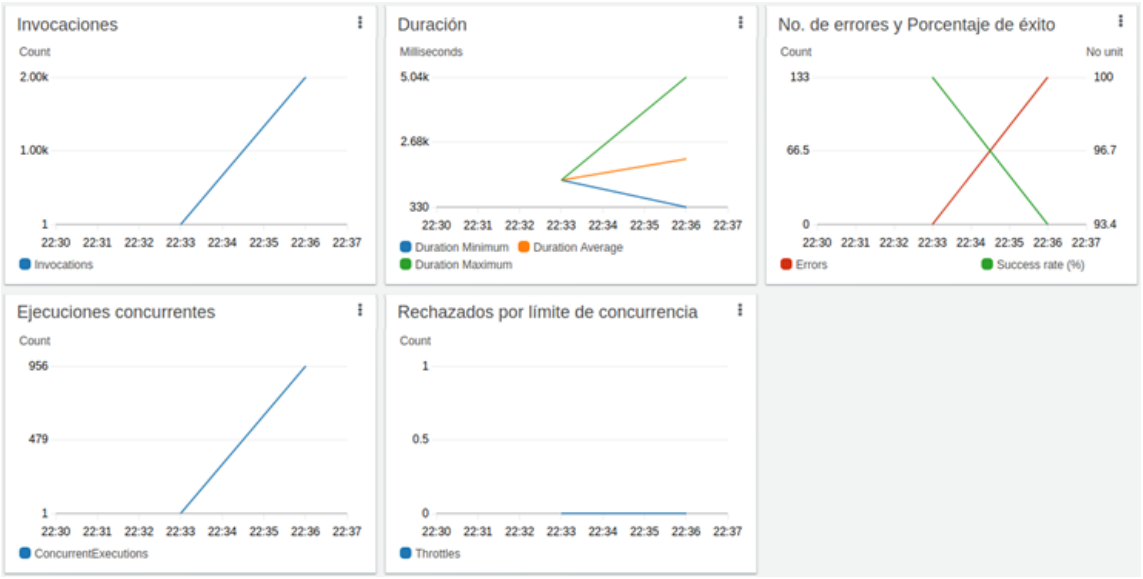
Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

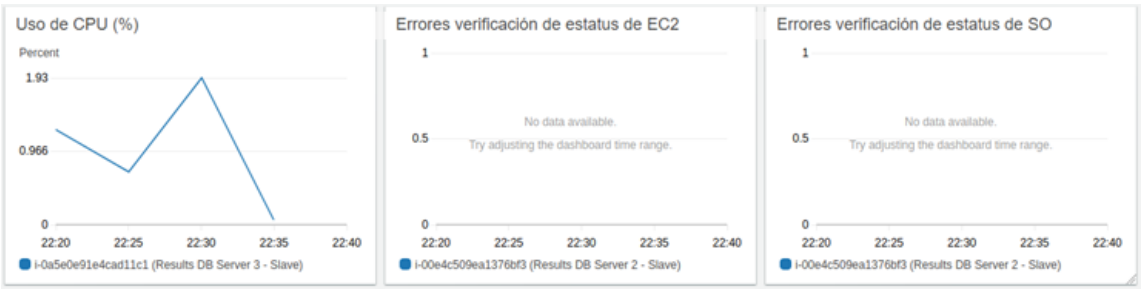
Continuación apéndice 9.

Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Figura C. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Aunque en algunas iteraciones se resolvieron exitosamente las 2000 solicitudes, fueron más veces en las que no se logró. Los 133 errores que reporta Artillery en la figura A fueron de tipo *timeout*, es decir, las solicitudes estaban demorando más de 5 segundos. Las métricas de la Lambda en la figura B reflejan los 133 errores de *timeout* que ocurrieron y las métricas de la instancia en la Figura C siguen mostrando un uso mínimo del procesador.

Escenario 8

Con el objetivo de mantener el *timeout* de la Lambda en 5 segundos, se optó por incrementar los recursos computacionales de la BD utilizando una instancia de memoria optimizada de tipo "*r6g.2xlarge*" con una capacidad de 64GB de memoria RAM. Luego de varias pruebas, se encontró que la configuración óptima de InnoDB para esta instancia es la que se muestra en la tabla siguiente.

Configuración de escenario 8

Descripción	2000 solicitudes entre 20 tareas en AWS Fargate.
Instancia EC2*	
Tipo	r6g.2xlarge
vCPUs	8
Memoria RAM	64GB
MySQL JS	
connectionLimit	2000
MySQL	

Continuación apéndice 9.

max_connections	2000
max_connect_errors	1000
innodb_buffer_pool_size*	58GB => 62277025792 B
innodb_buffer_pool_chunk_size*	2GB => 2147483648 B
innodb_buffer_pool_instances*	29
Lambda	
Timeout	5s
Reserved concurreny	1000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Figura A. **Resultados de Artillery**

Summary

Test duration	10 sec
Scenarios created	2000
Scenarios completed	2000

Scenario counts:

Request for results	2000 (100%)

Codes

200	2000

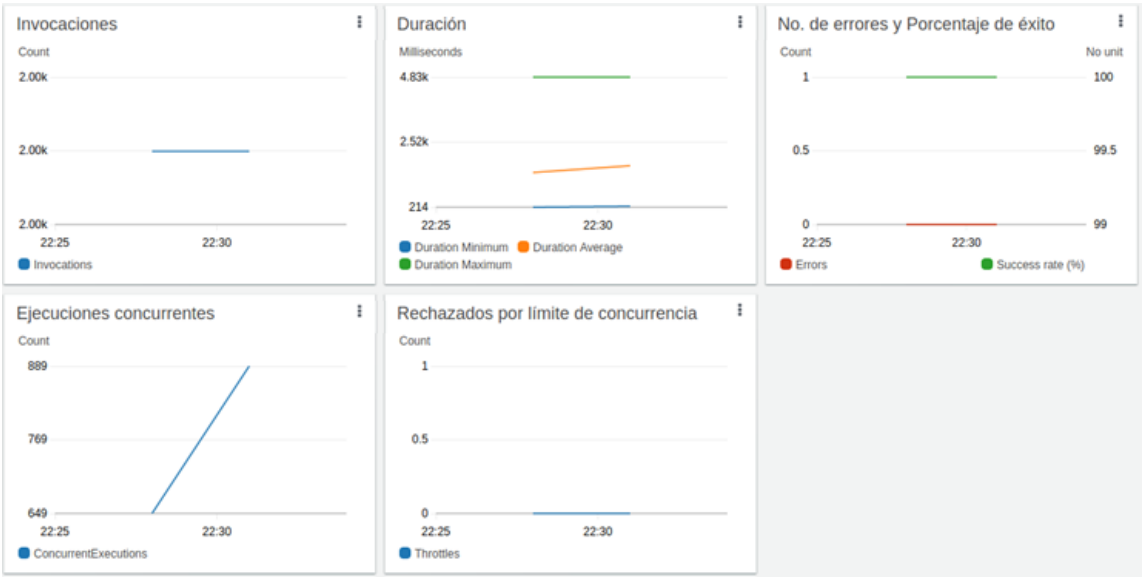
Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

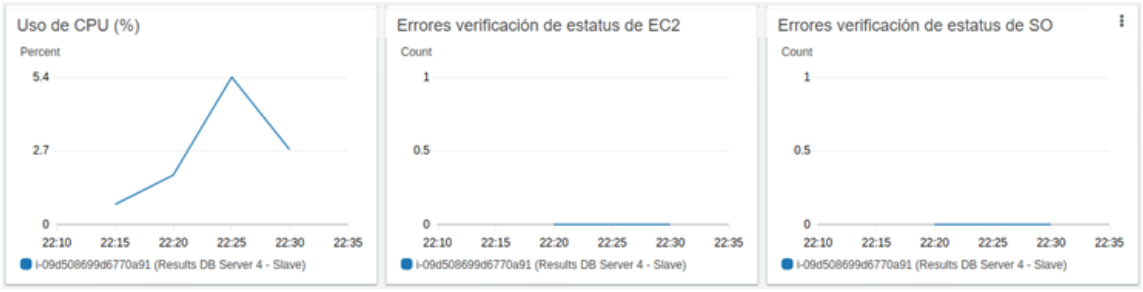
Continuación apéndice 9.

Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Figura C. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Los resultados de Artillery en muestran que se resolvieron correctamente el 100 % de las solicitudes. Al mismo tiempo, las métricas de la Lambda en la figura B indican que el tiempo máximo de ejecución fue de ~4.8 segundos y el gráfico de uso del procesador en la figura C indica que se alcanzó apenas un 5.4 % de la capacidad total. Aunque esta configuración podría servir para realizar la prueba completa de 2000 solicitudes a cada 10 segundos durante 5 minutos, está claro que en este punto el problema no es la cantidad de memoria RAM sino la optimización de MySQL. Por esta razón, se siguió experimentando con los parámetros de MySQL para encontrar una mejor configuración.

Escenario 9

Para entender mejor el funcionamiento de los *pools* de InnoDB se realizaron varias pruebas con la instancia "*t2.2xlarge*" variando los parámetros "*innodb_buffer_pool_chunk_size*" e "*innodb_buffer_pool_instances*". El objetivo era encontrar la configuración óptima de esta instancia ya que la "*r6g.2xlarge*" tiene un costo muy elevado por ser de memoria optimizada. Luego de probar todas las combinaciones posibles de estos dos parámetros para un *pool size* de 27GB, se encontró que a mayor número de instancias se obtenía un mejor resultado, de manera que 27 instancias de 1GB cada una resultó ser la configuración óptima tal como se muestra en la configuración de la tabla a continuación. No se muestra ningún gráfico de resultados, ya que en las 4 iteraciones con esta configuración se lograron resolver exitosamente las 2000 solicitudes.

Continuación apéndice 9.

Sin embargo, como se mencionó en la discusión de resultados, luego de revisar la documentación, el valor por defecto que sugiere MySQL es de 8 instancias, al igual como lo sugieren los autores del texto *High Performance MySQL*. Por esta razón, se consideró utilizar este mismo valor para el parámetro "*innodb_buffer_pool_instances*" para los escenarios siguientes.

Configuración de escenario 9

Descripción	2000 solicitudes entre 20 tareas en AWS Fargate.
Instancia EC2	
Tipo	t2.2xlarge
vCPUs	8
Memoria RAM	32GB
MySQL JS	
connectionLimit	2000
MySQL	
max_connections	2000
max_connect_errors	1000
innodb_buffer_pool_size	27GB => 28991029248 B
innodb_buffer_pool_chunk_size*	1GB => 1073741824 B
innodb_buffer_pool_instances*	27
Lambda	
Timeout	5s
Reserved concurreny	1000

Continuación apéndice 9.

Artillery	
Timeout	60s

Fuente: elaboración propia.

Escenario 10

Para utilizar 8 *pool instances* de InnoDB, fue necesario aumentar el tamaño del *pool* a 28GB para satisfacer la Ecuación 1. Como se observa en la tabla siguiente, el tamaño de cada instancia quedó en 3.5GB. Los resultados de Artillery que se muestran en la figura A corresponden a la primera iteración en el que ocurrieron 103 errores por *timeout*, pero luego se realizaron 4 iteraciones más y en las 4 se obtuvo un 100 % de éxito.

Configuración de escenario 10

Descripción	2000 solicitudes entre 20 tareas en AWS Fargate.
Instancia EC2	
Tipo	t2.2xlarge
vCPUs	8
Memoria RAM	32GB
MySQL JS	
connectionLimit	2000
MySQL	
max_connections	2000
max_connect_errors	1000

Continuación apéndice 9.

innodb_buffer_pool_size*	28GB => 30064771072 B
innodb_buffer_pool_chunk_size*	3.5GB => 3758096384 B
innodb_buffer_pool_instances*	8
Lambda	
Timeout	5s
Reserved concurreny	1000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Figura A. **Resultados de Artillery**

Summary

Test duration	10 sec
Scenarios created	2000
Scenarios completed	2000

Scenario counts:

Request for results	2000 (100%)

Codes

200	1897
500	103

Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

Continuación apéndice 9.

En general, la primera iteración de una prueba siempre genera más errores que las siguientes y la razón es que la primera vez que se ejecutan las funciones Lambda, tardan un tiempo en instanciar el contenedor y levantar el ambiente de ejecución. Las invocaciones siguientes se van a resolver más rápido porque se reutilizan las instancias anteriores.

Escenario 11

Con la configuración que se muestra en la tabla siguiente se realizó por primera vez la prueba completa de 5 minutos. Aparte de incrementar la concurrencia reservada de la Lambda a 2000, se introducen 2 nuevas variables de MySQL: "*innodb_read_io_threads*" y "*thread_cache_size*". De acuerdo con Schwartz *et al*., cuando se tiene una alta concurrencia de consultas a la BD, se puede mejorar el desempeño de las operaciones de lectura al disco duro incrementando este parámetro, cuyo valor por defecto es de 4. De preferencia, se recomienda utilizar un *thread* por unidad física de CPU. La segunda variable le indica a MySQL cuántos *threads* puede mantener en caché, de manera que cuando llega una nueva conexión, en vez de crear un nuevo *thread* para atender la solicitud, obtiene uno del caché lo cual tiene un potencial de mejorar el tiempo de respuesta cuando hay demasiadas conexiones. Se puede usar un valor tan alto como lo permita el Sistema Operativo y sin sobre pasar el límite por defecto de MySQL (Oracle Corporation, 2021).

Continuación apéndice 9.

Configuración de escenario 11

Descripción*	2000 solicitudes a cada 10 segundos durante 5 minutos distribuidos entre 20 tareas en AWS Fargate.
Instancia EC2	
Tipo	t2.2xlarge
vCPUs	8
Memoria RAM	32GB
MySQL JS	
connectionLimit	2000
MySQL	
max_connections	2000
max_connect_errors	1000
innodb_buffer_pool_size	28GB => 30064771072 B
innodb_buffer_pool_chunk_size	3.5GB => 3758096384 B
innodb_buffer_pool_instances	8
innodb_read_io_threads*	8
thread_cache_size*	200
Lambda	
Timeout	5s
Reserved concurreny*	2000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Continuación apéndice 9.

Figura A. **Resultados de Artillery**

Summary

Test duration	310 sec
Scenarios created	60000
Scenarios completed	60000

Scenario counts:

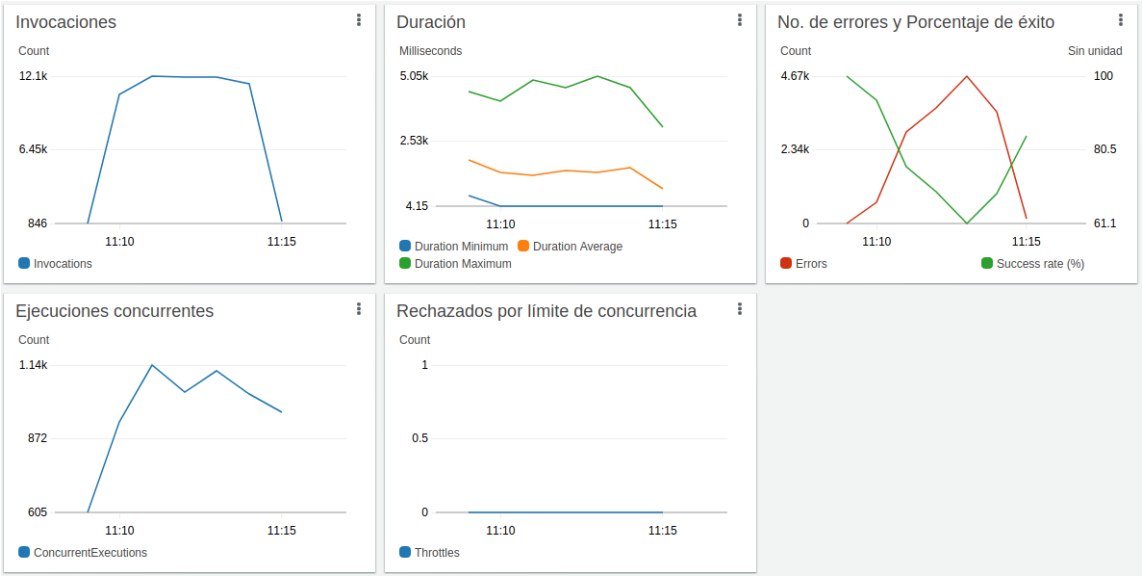
Get vote results	60000 (100%)

Codes

200	44378
500	15622

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

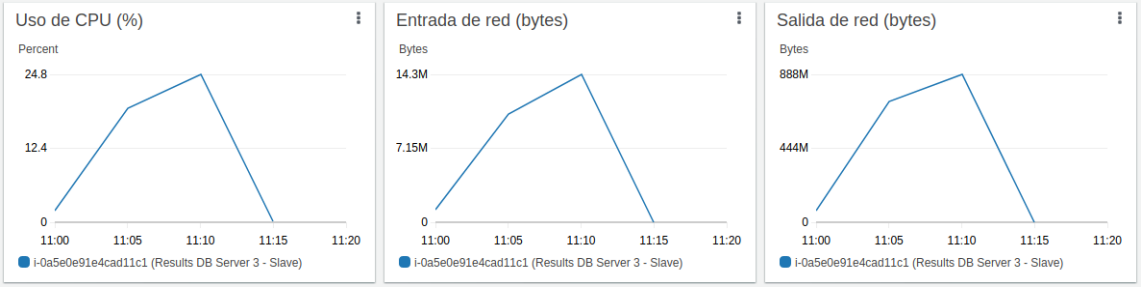
Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

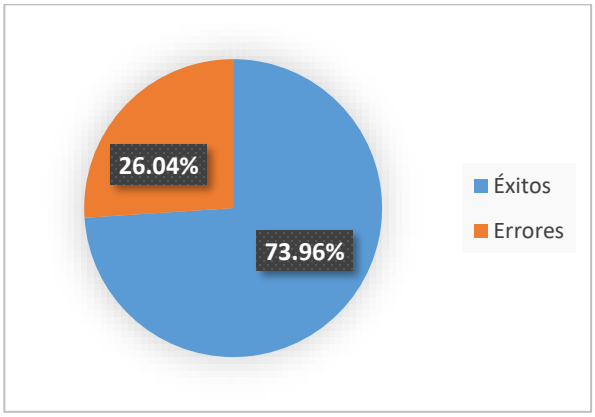
Continuación apéndice 9.

Figura C. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Figura D. **Lambda de resultados - Gráfico de disponibilidad**



Fuente: elaboración propia.

Utilizando la misma configuración del escenario 11 para la instancia "*t2.2xlarge*", los resultados de Artillery en la figura A indican que se resolvieron exitosamente 44,378 solicitudes de 60,000, logrando un nivel de disponibilidad del 73.96 % como lo muestra el gráfico de la figura D.

Continuación apéndice 9.

En las métricas de la Lambda en la figura B se puede observar que a pesar de que hubo muchos errores, no se rechazó ninguna solicitud. Los errores fueron causados por *timeouts* por lo que fue necesario aumentar este tiempo para lograr el objetivo. En la figura C, el gráfico de monitoreo del uso de CPU de la instancia EC2 de la BD indica que tuvo un uso máximo del 24.8 %, lo cual indica que ya se está aprovechando de mejor manera los recursos de procesamiento.

Escenario 12

Luego de realizar varias pruebas modificando la configuración del Escenario 11, se concluyó que la configuración óptima para la instancia "*t2.2xlarge*" es la que se muestra en la tabla a continuación. Como se puede notar, se aumentó el número máximo de conexiones que MySQL debe permitir a 9190 y el límite de concurrencia reservada para la Lambda se aumentó hasta 3000 ya que estaban rebotando muchas solicitudes. El número máximo de conexiones se había configurado en 10000 para aceptar todas las conexiones que fueran necesarias, sin embargo, MySQL lo corrigió automáticamente a 9190 debido a la capacidad de la instancia.

Continuación apéndice 9.

Configuración de escenario 12

Descripción	2000 solicitudes a cada 10 segundos durante 5 minutos distribuidos entre 20 tareas en AWS Fargate.
Instancia EC2	
Tipo	t2.2xlarge
vCPUs	8
Memoria RAM	32GB
MySQL JS	
connectionLimit*	3000
MySQL	
max_connections*	9190
max_connect_errors	1000
innodb_buffer_pool_size	28GB => 30064771072 B
innodb_buffer_pool_chunk_size	3.5GB => 3758096384 B
innodb_buffer_pool_instances	8
innodb_read_io_threads	8
thread_cache_size	200
Lambda	
Timeout	7s
Reserved concurreny*	3000
Artillery	
Timeout	60s

Fuente: elaboración propia.

Continuación apéndice 9.

Figura A. **Resultados de Artillery**

Summary

Test duration	310 sec
Scenarios created	60000
Scenarios completed	60000

Scenario counts:

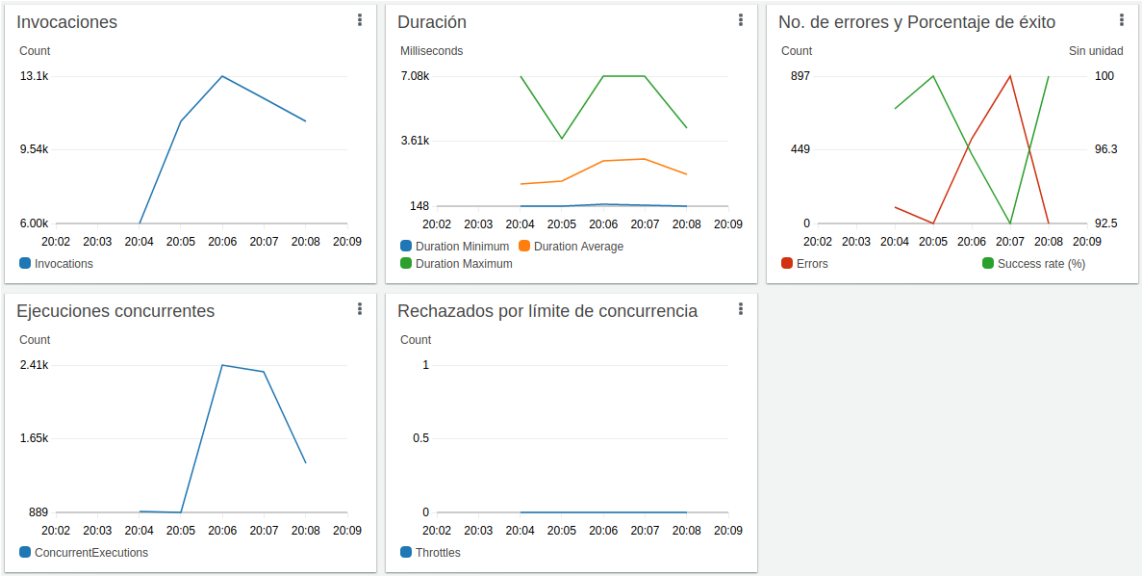
Get vote results	60000 (100%)

Codes

200	58486
500	1514

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

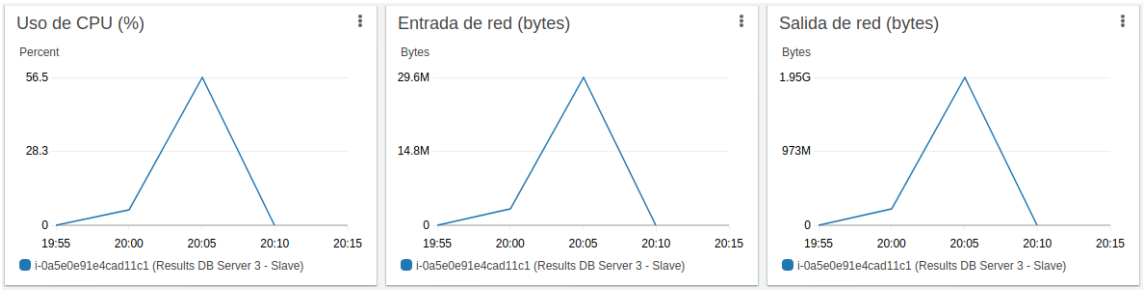
Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

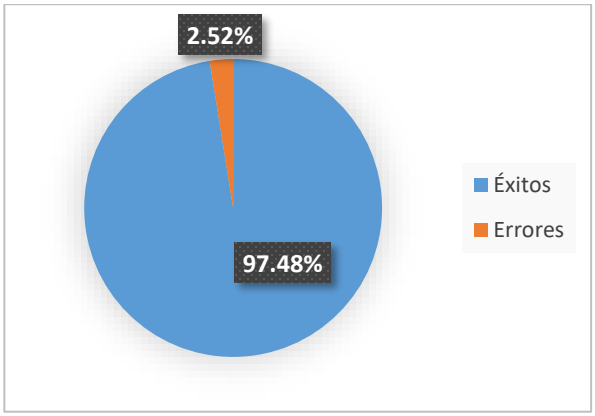
Continuación apéndice 9.

Figura C. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Figura D. **Lambda de resultados - Gráfico de disponibilidad**



Fuente: elaboración propia.

Con esta configuración, como puede observarse en los resultados de Artillery de la figura A, se logró resolver correctamente 58486 solicitudes y se obtuvo tan solo 1514 errores de *timeout*, generando un nivel de disponibilidad del 97.48 % como se muestra en el gráfico de la figura D, sin embargo, no alcanza el objetivo.

Continuación apéndice 9.

Como se puede observar en la figura B, no se rechazó ninguna solicitud y en cuanto al uso del procesador de la instancia, la figura C muestran que se alcanzó un nivel del 56.5 %, más del doble del valor que se obtuvo en el escenario 11.

Escenario 13

Luego de varias pruebas con la instancia de tipo "*t2.2xlarge*", se concluyó que no era capaz de resolver todas las solicitudes para obtener un 99.99 % de disponibilidad, por lo que finalmente se decidió migrar a la instancia "*c5.4xlarge*" que está optimizada para altas cargas de procesamiento, disponiendo de 16 vCPUs y 32GB de RAM. De nuevo se tuvieron que realizar varias iteraciones de pruebas para optimizar su configuración y el primer acercamiento fue intentar resolver exitosamente 2000 solicitudes a cada 10 segundos durante 3 minutos con un 100% de disponibilidad. Hacerlo durante 5 minutos era muy tardado y el costo de esta instancia es también muy elevado a comparación de la "*t2.2xlarge*". La siguiente tabla muestra la configuración completa de la prueba.

Configuración de escenario 13

Descripción*	2000 solicitudes a cada 10 segundos durante 3 minutos distribuidos entre 20 tareas en AWS Fargate.
Instancia EC2*	
Tipo	c5.4xlarge
vCPUs	16
Memoria RAM	32GB

Continuación apéndice 9.

MySQL JS	
connectionLimit*	7500
acquireTimeout*	12000ms
Query timeout*	12000ms
MySQL	
max_connections*	8000
max_connect_errors*	3000
innodb_buffer_pool_size	28GB => 30064771072 B
innodb_buffer_pool_chunk_size*	1.75GB => 1879048192 B
innodb_buffer_pool_instances*	16
innodb_read_io_threads	8
thread_cache_size*	250
Lambda	
Timeout	15s
Reserved concurreny*	3800
Artillery	
Timeout	60s

Fuente: elaboración propia.

Continuación apéndice 9.

Mientras se realizaron las pruebas iniciales con esta nueva instancia, se fueron descubriendo nuevas configuraciones. Para la librería de MySQL que se usa en la función de la Lambda para conectarse a la BD, se aumentó el límite de conexiones a 7500 ya que las solicitudes se acumulaban en cada iteración. También se incrementó a 12 segundos el tiempo de espera para obtener una nueva conexión (*acquireTimeout*), al igual que se hizo con el tiempo de espera para que una consulta sea resuelta por la BD (*query timeout*), ya que el valor por defecto de estos dos parámetros es de 10 segundos.

En la configuración de MySQL se estableció en 8000 el número máximo de conexiones, mientras que el número máximo de errores al intentar una conexión se subió a 3000 (*max_connect_errors*). En cuanto a InnoDB, se definieron 16 instancias de *pool* para aprovechar las 16 unidades de CPU que tiene la instancia. El número máximo de *threads* en caché se estableció en 250, luego de varias pruebas variando este parámetro. Por último, el límite de concurrencia reservada para la Lambda se aumentó hasta 3800 ya que muchas solicitudes se quedaban en cola y cuando llegaba la siguiente iteración, muchas terminaban rebotando.

Continuación apéndice 9.

Figura A. **Resultados de Artillery**

Summary

Test duration	190 sec
Scenarios created	36000
Scenarios completed	36000

Scenario counts:

Get vote results	36000 (100%)

Codes

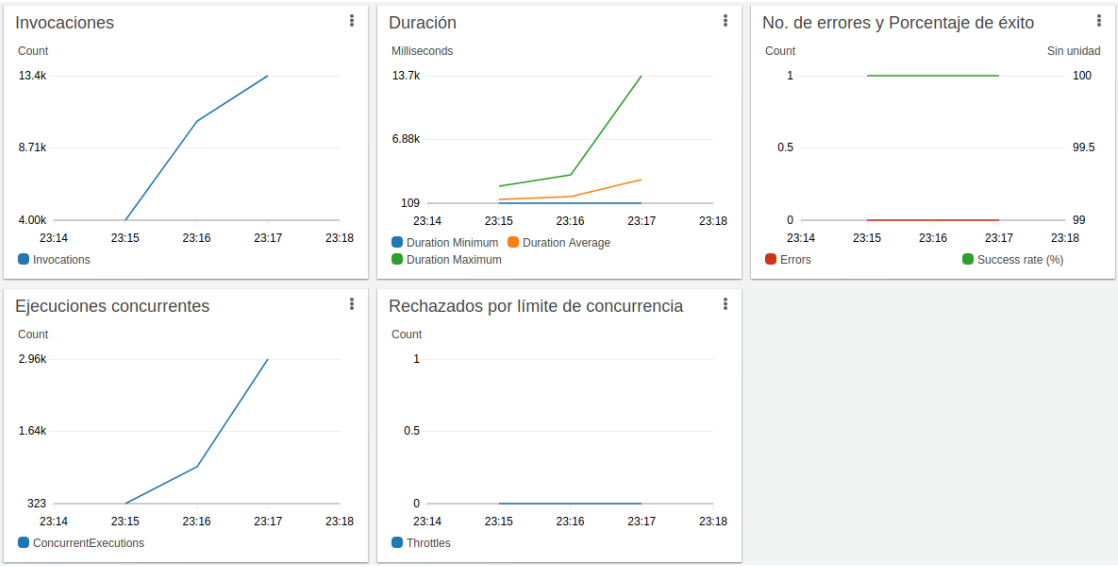
200	36000

Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

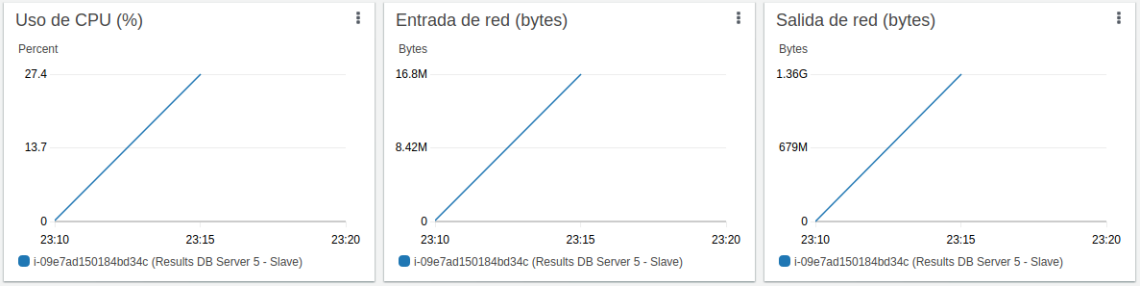
Figura B. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Figura C. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

La figura A muestra que con esta configuración optimizada se logró resolver el 100 % de las solicitudes que se generaron en 3 minutos. Las métricas de la Lambda en la figura B no muestran ningún error, aunque la duración máxima de ejecución alcanzó hasta ~13.7 segundos. De acuerdo con la figura C, se utilizó únicamente un 27.4 % de la capacidad total del procesador.

Escenario 14

Los cambios más importantes en este escenario respecto del escenario 13 radican en la configuración de MySQL. Como se puede observar en la tabla de configuración, se tuvo que maximizar los parámetros de MySQL para obtener la menor cantidad de errores posibles. Se aumentó hasta 8100 el límite de conexiones y hasta 4000 el número de intentos de conexión fallidos (*max_connect_errors*). Después de varios experimentos, se verificó que 32 instancias del *pool* de 0.875GB cada uno optimizan el rendimiento del *buffer* de InnoDB.

Continuación apéndice 9.

Además, se utilizó el máximo número de concurrencia reservada para la Lambda de resultados, tomando en cuenta que el límite era de 4000, luego de haber solicitado un incremento a AWS. Debido a que siempre se deben dejar 100 unidades libres, el valor máximo que se podía usar era de 3900. Sin embargo, la Lambda para consultar la información pública está activa, por lo que se le asignó un número mínimo de 10 concurrencias reservadas, dejando disponibles 3890 unidades para la Lambda de resultados.

Configuración de escenario 14

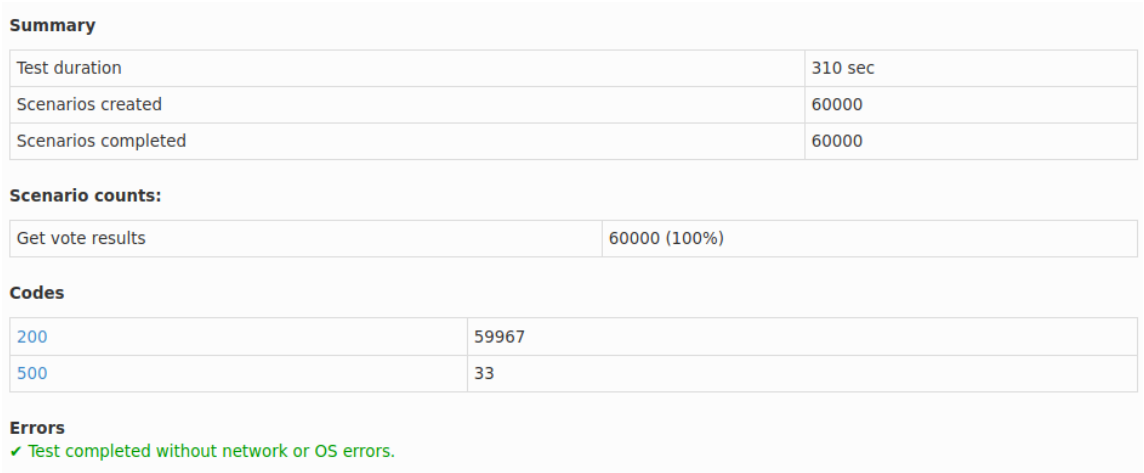
Descripción*	2000 solicitudes a cada 10 segundos durante 5 minutos distribuidos entre 20 tareas en AWS Fargate.
Instancia EC2	
Tipo	c5.4xlarge
vCPUs	16
Memoria RAM	32GB
MySQL JS	
connectionLimit*	8000
acquireTimeout*	17000ms
Query timeout*	17000ms
MySQL	
max_connections*	8100
max_connect_errors*	4000
innodb_buffer_pool_size	28GB => 30064771072 B
innodb_buffer_pool_chunk_size*	0.875GB => 939524096 B
innodb_buffer_pool_instances*	32

Continuación apéndice 9.

innodb_read_io_threads	8
thread_cache_size	250
Lambda	
Timeout*	18s
Reserved concurreny*	3890
Artillery	
Timeout	60s

Fuente: elaboración propia.

Figura A. **Resultados de Artillery**



Summary

Test duration	310 sec
Scenarios created	60000
Scenarios completed	60000

Scenario counts:

Get vote results	60000 (100%)

Codes

200	59967
500	33

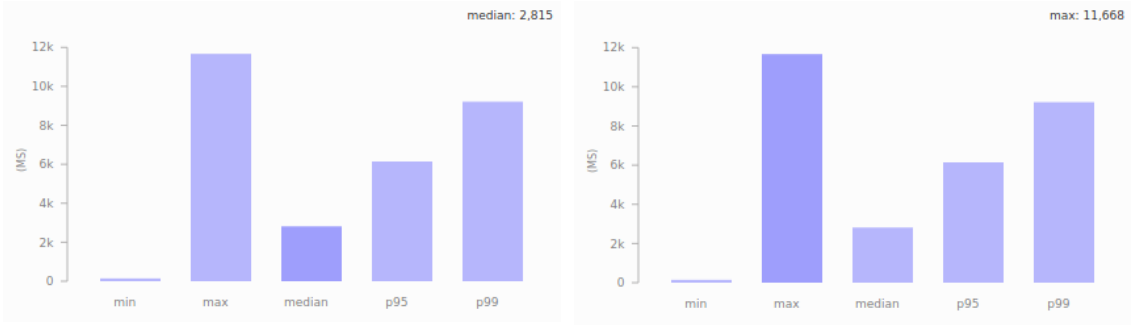
Errors

✔ Test completed without network or OS errors.

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

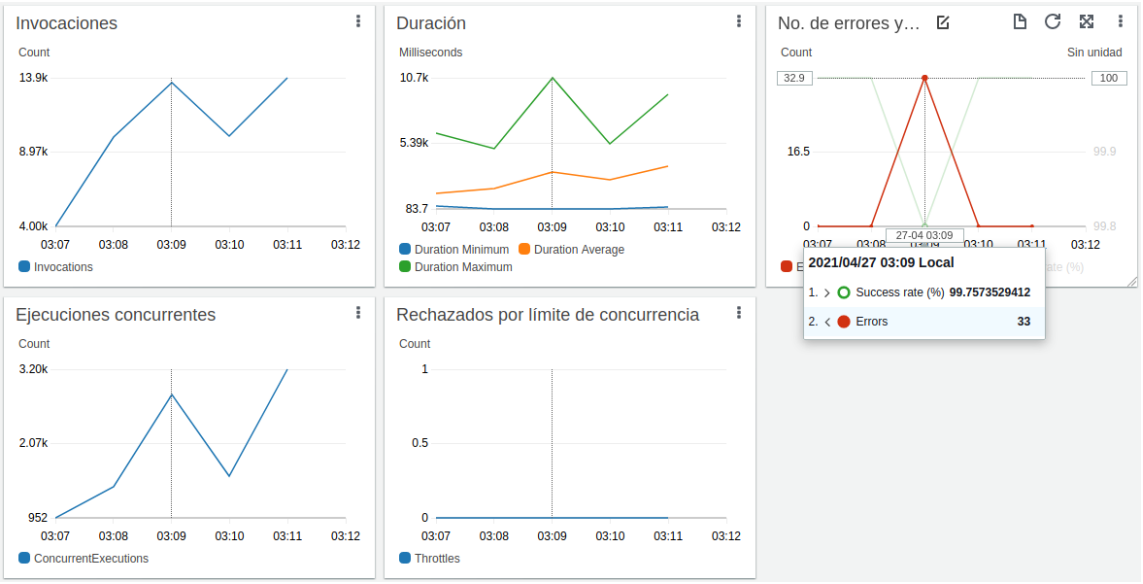
Continuación apéndice 9.

Figura B. **Distribución de latencia**



Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

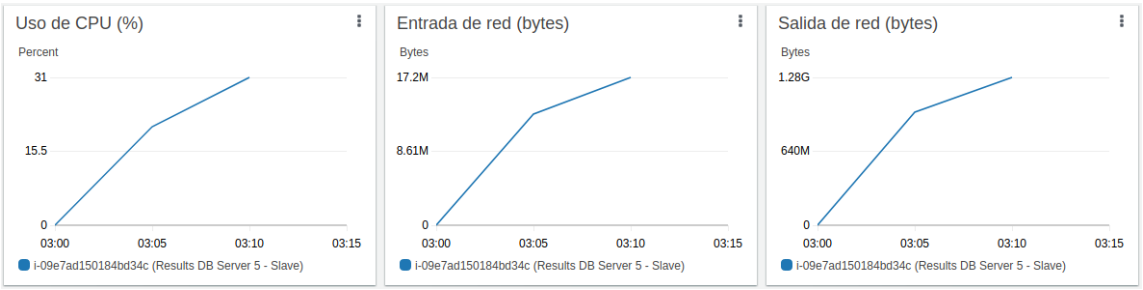
Figura C. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

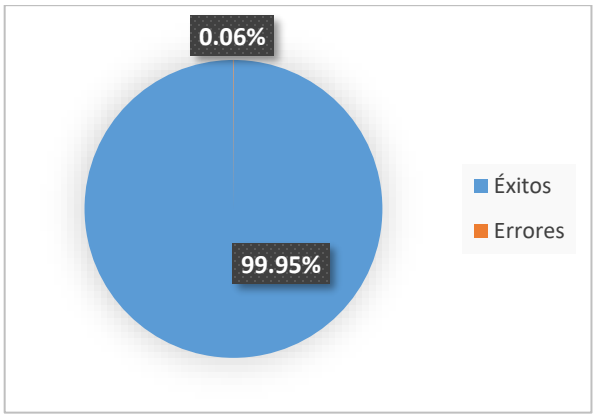
Continuación apéndice 9.

Figura D. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Figura E. **Gráfico de disponibilidad**



Fuente: elaboración propia.

Continuación apéndice 9.

De acuerdo con los resultados de Artillery en la figura A, con esta configuración se logró resolver 59,967 solicitudes y tan solo se generaron 33 errores, logrando un nivel de disponibilidad del 99.95 % como se muestra en el gráfico de pastel de la figura E. Dado que con este rendimiento se alcanzó el objetivo del proyecto, se decidió utilizar los resultados de esta prueba como los resultados finales.

El gráfico de errores en la figura C muestra justamente los 33 errores que reportó Artillery, los cuáles fueron generados por *timeout* de la Lambda, es decir, la instancia de la BD no logró resolverlos en menos de 18 segundos. Artillery genera también una gráfica de la distribución de los tiempos de latencia, tal como se observa en la figura B y se habla de ello en la discusión de los resultados. El gráfico de porcentaje de uso del procesador en la figura D muestra que se alcanzó hasta un 31 % de la capacidad, por lo que la instancia no fue sobre cargada.

Escenario 15

Esta prueba se realizó únicamente con el objetivo de demostrar que, al tener un servidor de BD mucho más potente, se puede garantizar una disponibilidad del 100 %. La siguiente instancia en la familia “*c5*”, es la “*c5.9xlarge*” que ofrece 36 unidades de procesador y 72GB de memoria RAM. Para ello, se hizo una solicitud a AWS para incrementar hasta 36 el número de vCPUs que se podían usar en la cuenta, ya que el límite por defecto es de 32 vCPUs.

Continuación apéndice 9.

Como se observa en la siguiente tabla de configuración, aparte de usar una instancia con más recursos, también se modificaron algunos parámetros de MySQL para sacarle el máximo provecho a la instancia "*c5.9xlarge*". Se utilizó 68GB de los 72GB disponibles de memoria RAM para el *buffer* de InnoDB y se crearon 32 instancias del *pool* para aprovechar los 36 vCPUs que ofrece la instancia. También se aumentó el valor de "*innodb_read_io_threads*" a 16, es decir, 1 por cada núcleo de procesador físico.

Además, esta instancia permitió reducir el número máximo de conexiones a 4000 y el número máximo de intentos de conexión fallidos a 1000. Lo más importante es que se redujo el *timeout* de la Lambda a solo 10 segundos ya que, al tener más núcleos de procesador, se lograron resolver más solicitudes de forma concurrente.

Configuración de escenario 15

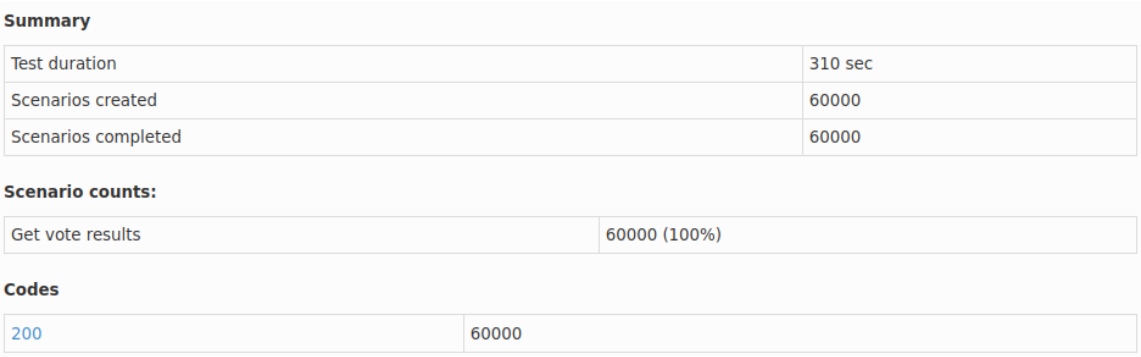
Descripción	2000 solicitudes a cada 10 segundos durante 5 minutos distribuidos entre 20 tareas en AWS Fargate.
Instancia EC2*	
Tipo	c5.9xlarge
vCPUs	36
Memoria RAM	72GB
MySQL JS	
connectionLimit	8000
acquireTimeout	17000ms
Query timeout	17000ms

Continuación apéndice 9.

MySQL	
max_connections*	4000
max_connect_errors*	1000
innodb_buffer_pool_size	68GB => 73014444032 B
innodb_buffer_pool_chunk_size*	2.125GB => 2281701376 B
innodb_buffer_pool_instances*	32
innodb_read_io_threads*	16
thread_cache_size	250
Lambda	
Timeout*	10s
Reserved concurreny	3890
Artillery	
Timeout	60s

Fuente: elaboración propia.

Figura A. **Resultados de Artillery**



Summary

Test duration	310 sec
Scenarios created	60000
Scenarios completed	60000

Scenario counts:

Get vote results	60000 (100%)

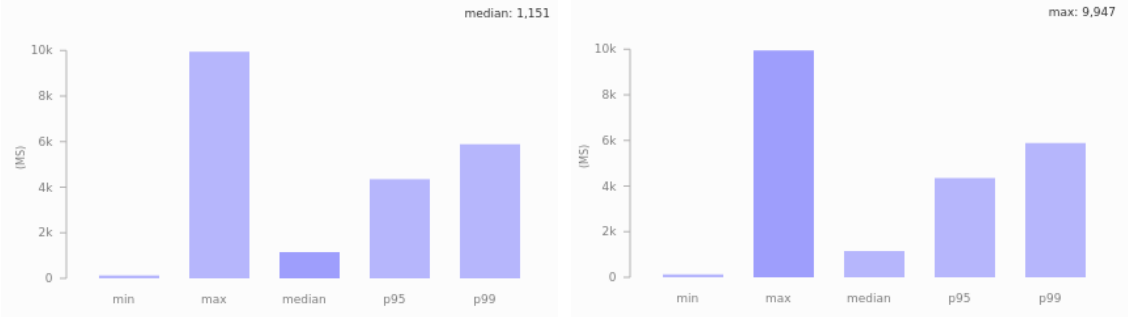
Codes

200	60000

Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

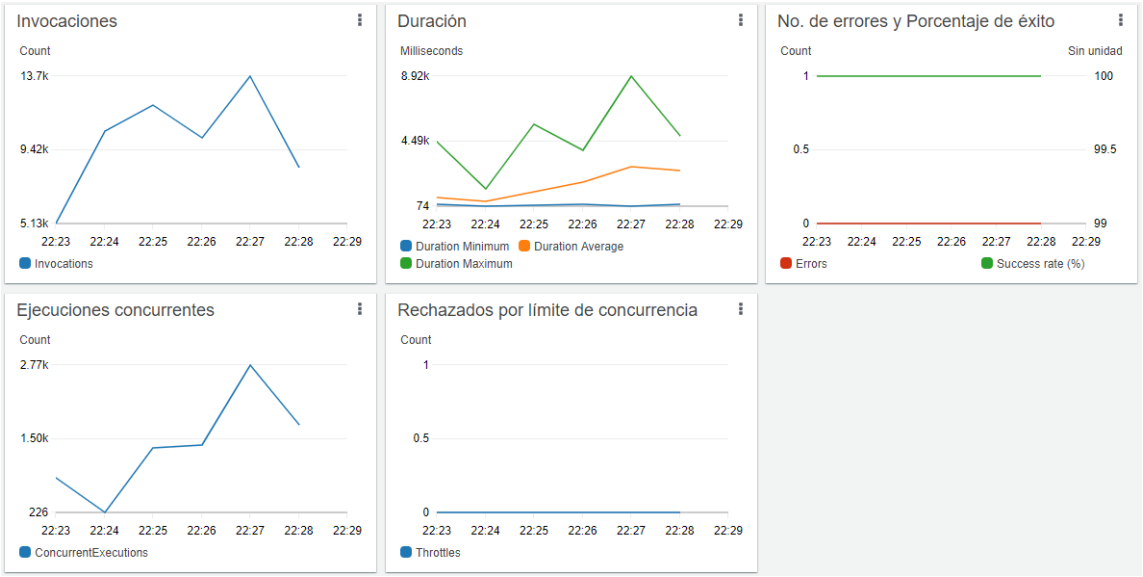
Continuación apéndice 9.

Figura B. **Distribución de latencia**



Fuente: elaboración propia, utilizando comando de Artillery para generar reportes.

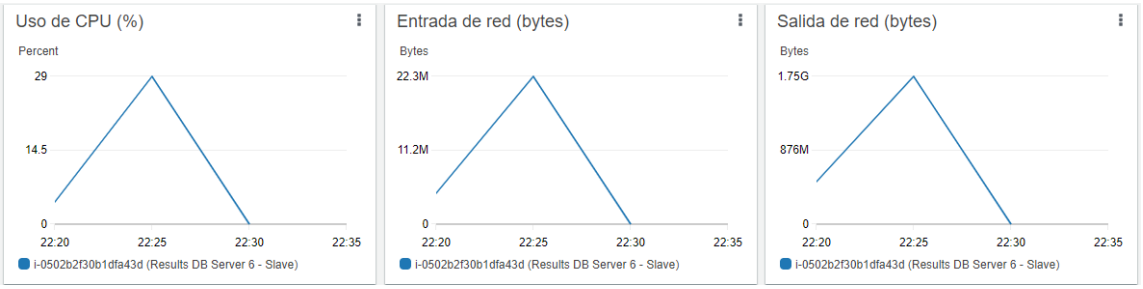
Figura C. **Métricas de la Lambda**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Continuación apéndice 9.

Figura D. **Monitoreo de EC2**



Fuente: elaboración propia, utilizando consola de Amazon CloudWatch.

Como se observa en la figura A, los resultados de Artillery muestran que se logró resolver el 100 % de las solicitudes y esto se puede confirmar con el gráfico de errores en las métricas de la Lambda en la figura C. El gráfico de duración de la Lambda también indica que la duración máxima fue alrededor de ~9 segundos mientras la duración promedio se mantuvo menor a 4 segundos. Por último, el gráfico del porcentaje de uso de CPU en la figura D indica que se alcanzó apenas un uso máximo del 29 %.

Fuente: elaboración propia.

Apéndice 10. **Alternativa de arquitectura usando ALB de AWS**

A continuación, se propone una arquitectura que permite escalar de forma horizontal, a diferencia de la arquitectura que se propuso en este proyecto que escala verticalmente. Esta arquitectura se basa principalmente en el uso de Balanceadores de Carga a nivel de Aplicación (*Application Load Balancer o ALB*) para distribuir la carga de solicitudes entre varias instancias de EC2. De igual manera, utiliza una arquitectura de replicación *Master-Slave* para tener varias BDs secundarias donde se van a realizar las operaciones de lectura.

Un ALB permite especificar reglas de ruteo y cuando llega una solicitud, verifica la regla de ruteo que aplica y redirecciona las solicitudes al grupo objetivo (*target group*) de la regla que determinó, el cual puede ser un conjunto de instancias de EC2 o una Lambda. Existen varias formas de definir las reglas de ruteo, como evaluar puertos (80, 443), patrones de URLs, rangos de IPs, evaluar un *header* de la solicitud HTTP, entre otros (Amazon Web Services, Inc., 2021).

Un ALB tiene la limitación de que solo permite usar una Lambda como "*target group*" de una regla por lo que, para usar varias Lambdas habría que definir varias reglas de ruteo y se complica más la implementación. Además, también tiene la limitación de que la Lambda no debe devolver más de 1MB de data (Amazon Web Services, Inc., 2021).

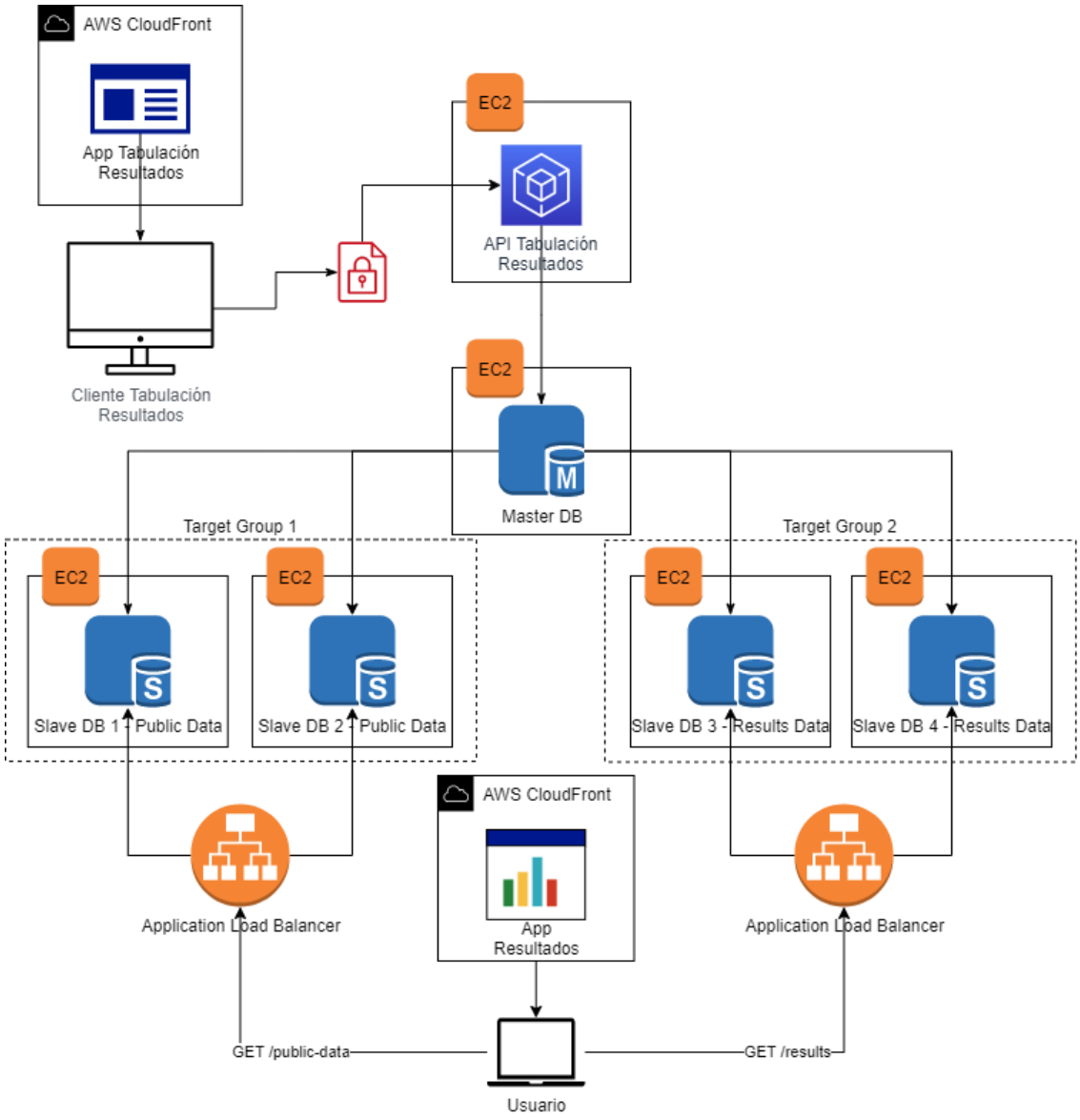
Continuación apéndice 10.

Utilizar instancias de EC2 como “*target group*” del ALB, permite especificar varias instancias por lo que la regla de ruteo sería tan simple como redirigir todo el tráfico HTTPS (puerto 443) a un grupo objetivo, que a su vez tiene asociado dos o más instancias de EC2. De esta manera, el ALB distribuye de forma inteligente el tráfico entre las instancias de EC2 y como consecuencia, las BDs secundarias van a recibir una menor concurrencia de consultas.

La arquitectura usa dos ALB para facilitar la tarea: una para la consulta de resultados y otra para la consulta de la información pública. Sin embargo, es posible usar un solo ALB y definir dos reglas de ruteo en el ALB basados en la URL: 1) Redirigir todas las solicitudes de “*/results*” al “*Target 2*”, y 2) Redirigir todas las solicitudes de “*/public-data*” al “*Target 1*”.

Continuación apéndice 10.

Arquitectura en AWS usando un balanceador de carga



Fuente: elaboración propia, utilizando diagrams.net.