# Universidad Nacional de Loja

## Facultad de la Energía, las Industrias y los de Recursos Naturales No Renovables

### Maestría en Ingeniería en Software

# Escalamiento Automático en un Sistema de Gestión de Reservas con Alta Disponibilidad Usando Azure

**Trabajo de Titulación, previo a la obtención del título de Magíster en Ingeniería en Software**


**AUTOR**:

Omar Alexis Sanmartin Tapia

**DIRECTOR**:

Ing. Roberth Gustavo Figueroa Díaz, Mg. Sc.


**Loja – Ecuador**

**2025**

**Certificación**

Loja, 26 de febrero del 2025

Ing. Roberth Gustavo Figueroa Díaz, Mg. Sc.
**DIRECTOR DEL TRABAJO DE TITULACIÓN**

**CERTIFICO:**

Que he revisado y orientado todo el proceso de elaboración del Trabajo de Titulación denominado: **Escalamiento Automático en un Sistema de Gestión de Reservas con Alta Disponibilidad Usando Azure**, previo a la obtención del título de **Magíster en Ingeniería en Software**, de la autoría del estudiante **Omar Alexis Sanmartin Tapia** con cedula de identidad **Nro. 1105381014**, una vez que el trabajo cumple con todos los requisitos exigidos por la Universidad Nacional de Loja, para el efecto, autorizo la presentación del mismo para su respectiva sustentación y defensa.

Firmado electrónicamente por:
ROBERTH GUSTAVO
FIGUEROA DIAZ

Ing. Roberth Gustavo Figueroa Díaz, Mg. Sc.

**DIRECTOR DEL TRABAJO DE TITULACIÓN**

## Autoría

Yo, **Omar Alexis Sanmartin Tapia** declaro ser autor del presente Trabajo de Titulación y eximo expresamente a la Universidad Nacional de Loja y a sus representantes jurídicos de posibles reclamos o acciones legales por el contenido del mismo. Adicionalmente, acepto y autorizo a la Universidad Nacional de Loja, la publicación del Trabajo de Titulación, en el Repositorio Digital Institucional – Biblioteca Virtual.


**Firma:**

**Cédula de identidad:** 1105381014

**Fecha:** 26 de febrero de 2025.

**Correo electrónico:** omar.sanmartin@unl.edu.ec

**Teléfono:** 0981019216

**Carta de autorización por parte del autor, para la consulta, reproducción parcial o total y/o publicación electrónica del texto completo del Trabajo de Titulación.**

Yo, **Omar Alexis Sanmartin Tapia,** declaro ser autor del Trabajo de Titulación denominado: **Escalamiento Automático en un Sistema de Gestión de Reservas con Alta Disponibilidad Usando Azure**, como requisito para optar al título de **Magíster en Ingeniería en Software,** autorizo al sistema Bibliotecario de la Universidad Nacional de Loja para que, con fines académicos, muestre al mundo la producción intelectual de la Universidad, a través de la visibilidad de su contenido en el Repositorio Digital Institucional:

Los usuarios pueden consultar el contenido de este trabajo en el Repositorio Institucional, en las redes de información del país y del exterior, con los cuales tenga convenio la Universidad.

La Universidad Nacional de Loja, no se responsabiliza por plagio o copia del Trabajo de Titulación que realice un tercero.

Para constancia de esta autorización, suscribo en la ciudad de Loja, a los veinte seis días del mes de febrero de dos mil veinte cinco.

Firmado electrónicamente por:
OMAR ALEXIS
SANMARTIN TAPIA

**Firma:**

**Autor:** Omar Alexis Sanmartin Tapia

**Cédula:** 1105381014

**Dirección:** Barrio Primavera, Catamayo - Ecuador

**Correo electrónico:** omar.sanmartin@unl.edu.ec

**Teléfono:** 0981019216

**DATOS COMPLEMENTARIOS**

**Director del Trabajo de Titulación:** Ing. Roberth Gustavo Figueroa Díaz, Mg. Sc.

## Dedicatoria

Este trabajo está dedicado a toda mi familia: Jenny, Nelly, Dome, y Johanna, por su infinita paciencia, apoyo incondicional y la dedicación que me han brindado en cada paso de este camino. Gracias por ser mi inspiración, por inculcarme valores sólidos y un ejemplo a seguir, y por demostrarme que los sueños se alcanzan con esfuerzo y perseverancia.

A ustedes, quienes nunca se dieron por vencidos, incluso en los momentos más difíciles, y me recordaron siempre que el éxito es el resultado del trabajo constante y el amor por lo que hacemos. Este logro no solo es mío, sino también de ustedes.

Con todo mi cariño y gratitud, este esfuerzo es para ustedes.

***Omar Alexis Sanmartin Tapia***

**Agradecimiento**

Quiero expresar mi más profunda gratitud a Dios, cuya bendición y guía iluminan siempre mi vida, dándome fortaleza y esperanza en cada paso de este camino.

A mi familia, por estar siempre presente, brindándome su apoyo incondicional, su amor y sus palabras de aliento que han sido mi motor para seguir adelante.

A Johanna Montaño, mi fiel amiga, quien creyó en mí incluso antes de que comenzara este viaje, y cuya confianza y apoyo han sido un pilar fundamental en mi vida.

Mi sincero agradecimiento a todos los docentes, por compartir su conocimiento y sabiduría, a mis compañeros, por los aprendizajes compartidos, y a mis amigos, por ser una fuente de inspiración y apoyo constante. Gracias a todos ustedes por ayudarme a crecer, no solo profesionalmente, sino también como persona, a lo largo de esta etapa tan significativa de mi vida.

***Omar Alexis Sanmartin Tapia***

## Índice de Contenidos

## Índice de tablas

## Índice de figuras

## Índice de anexos

## Glosario de términos y Acrónimos

- **Alta Disponibilidad (High Availability):** Capacidad de un sistema para permanecer operativo y accesible durante la mayor parte del tiempo, minimizando los tiempos de inactividad.
- **Azure Functions:** Servicio de computación en la nube que permite ejecutar pequeñas piezas de código (funciones) sin necesidad de administrar servidores. Es una solución ideal para arquitecturas serverless.
- **Base de Datos Relacional:** Tipo de base de datos estructurada que organiza datos en tablas relacionadas entre sí mediante claves primarias y externas.
- **Cold Start (Inicio en Frío):** Tiempo que toma un servicio serverless para iniciar desde un estado inactivo. Es un factor importante en el rendimiento de aplicaciones serverless.
- **Concurrencia:** Capacidad de un sistema para manejar múltiples solicitudes o procesos simultáneamente.
- **DevOps:** Conjunto de prácticas y herramientas que unifican el desarrollo (Dev) y las operaciones (Ops) para acelerar la entrega de software y mejorar su calidad.
- **Escalamiento Automático:** Mecanismo que ajusta automáticamente los recursos asignados a una aplicación en función de la carga de trabajo, aumentando o disminuyendo los recursos según sea necesario.
- **Event-driven (Basado en Eventos):** Modelo de programación donde las operaciones se activan en respuesta a eventos, como solicitudes HTTP o cambios en una base de datos.
- **JSON (JavaScript Object Notation):** Formato ligero para el intercambio de datos, ampliamente utilizado en aplicaciones web para enviar y recibir datos entre el cliente y el servidor.
- **Microservicios:** Arquitectura de software que divide una aplicación en pequeños servicios independientes, cada uno con una funcionalidad específica.
- **Node.js:** Entorno de ejecución de JavaScript basado en el motor V8 de Google Chrome. Es utilizado para construir aplicaciones escalables y rápidas, especialmente en entornos serverless.
- **ORM (Object-Relational Mapping):** Técnica para interactuar con bases de datos relacionales usando objetos en el código. Sequelize es un ejemplo popular de ORM en Node.js.
- **REST (Representational State Transfer):** Estilo arquitectónico para servicios web que utiliza HTTP para interactuar con recursos a través de métodos estándar como GET, POST, PUT y DELETE.

- **Serverless:** Modelo de computación en la nube donde los desarrolladores no administran servidores directamente, sino que implementan funciones o servicios que se escalan automáticamente.
- **V8:** Motor de JavaScript de alto rendimiento desarrollado por Google, utilizado por Node.js para ejecutar código JavaScript en el lado del servidor.
- **Variable de Entorno:** Parámetros que configuran el comportamiento de una aplicación en diferentes entornos (desarrollo, prueba, producción). Usadas comúnmente para almacenar configuraciones sensibles como claves de acceso.

## 1. Titulo

**Escalamiento Automático en un Sistema de Gestión de Reservas con Alta Disponibilidad Usando Azure.**

## 2. Resumen

Este trabajo de titulación aborda la implementación de un sistema de gestión de reservas utilizando una arquitectura serverless en Azure Functions para garantizar alta disponibilidad y escalabilidad. Inicialmente, el sistema fue desarrollado en un modelo basado en servidor utilizando Express.js, pero debido a los objetivos del proyecto, se migró a una arquitectura serverless, implicando un refactor significativo del código para adaptarlo a las características de este entorno.

El proyecto incluyó la configuración de servicios serverless, la integración con sistemas de escalamiento automático y la ejecución de pruebas de carga y estrés para evaluar su comportamiento bajo distintas condiciones. Durante las pruebas, se observó que los servicios no alcanzaron un escalamiento efectivo, con un uso máximo de CPU de 9% en AuthCore y 41% en BookingCore. Adicionalmente, se identificaron cuellos de botella en la base de datos, que resultaron en errores frecuentes de conexión y tiempos de respuesta elevados en escenarios de alta carga.

Este trabajo concluye que, si bien las arquitecturas serverless ofrecen una base prometedora para aplicaciones críticas, su desempeño depende directamente de la configuración y optimización de la infraestructura de soporte, como el manejo de conexiones de base de datos y el uso de mecanismos de caché para reducir la dependencia de operaciones intensivas en recursos.

**Palabras Clave:** Azure, Function App, Node.js, Serverless, Jmeter

**Abstract**

This degree work addresses the implementation of a reservation management system using a serverless architecture in Azure Functions to ensure high availability and scalability. Initially, the system was developed in a server-based model using Express.js, but due to the objectives of the project, it was migrated to a serverless architecture, involving a significant refactoring of the code to adapt it to the characteristics of this environment.

The project included the configuration of serverless services, the integration with automatic scaling systems and the execution of load and stress tests to evaluate its behavior under different conditions. During the tests, it was observed that the services did not scale effectively, with a maximum CPU usage of 9% in AuthCore and 41% in BookingCore. Additionally, database bottlenecks were identified, resulting in frequent connection errors and high response times in high load scenarios.

This paper concludes that while serverless architectures offer a promising foundation for critical applications, their performance is directly dependent on the configuration and optimization of the supporting infrastructure, such as database connection handling and the use of caching mechanisms to reduce dependency on resource-intensive operations.

**Keywords:** Azure, Function App, Node.js, Serverless, Jmeter

## 3. Introducción

La Alta Disponibilidad (AD) es un concepto fundamental en el diseño de sistemas informáticos críticos, garantizando la continuidad operativa y la mínima interrupción en los servicios, incluso bajo condiciones de carga elevada o fallos inesperados. Este trabajo se enfoca en investigar e implementar arquitecturas de software que ofrezcan Alta Disponibilidad mediante el uso de tecnologías modernas como Kubernetes, Node.js y Azure Cloud, evaluando su efectividad en escenarios reales.

El caso de estudio se centra en un sistema de reservas, un entorno ideal para evaluar las capacidades de una arquitectura de Alta Disponibilidad debido a sus demandas fluctuantes y su naturaleza crítica. En este contexto, los sistemas de reservas suelen enfrentar desafíos como picos de tráfico inesperados, tiempos de inactividad significativos y complejidad en la gestión de recursos. Estos retos subrayan la importancia de adoptar arquitecturas diseñadas para escalar y garantizar la disponibilidad continua.

La relevancia de este estudio radica en su potencial para ofrecer soluciones replicables que combinen escalabilidad y resiliencia, contribuyendo tanto al ámbito académico como a las aplicaciones prácticas en diversas industrias. Estudios previos destacan la eficacia de Kubernetes y Azure para manejar cargas dinámicas y garantizar la disponibilidad de servicios críticos [1][2], lo que refuerza la pertinencia de esta investigación.

El objetivo principal de este trabajo es diseñar e implementar una arquitectura que minimice los tiempos de inactividad y garantice la alta disponibilidad en un sistema de reservas. Entre los objetivos secundarios se incluyen la validación del desempeño del sistema en condiciones de carga elevada y la integración de herramientas de monitoreo para asegurar la detección temprana de fallos.

Los alcances de este trabajo se limitan al desarrollo de un prototipo funcional y su evaluación en un entorno controlado. Sin embargo, el análisis no incluye una implementación a gran escala ni pruebas en entornos de producción, debido a restricciones de tiempo y recursos.

# 4. Marco Teórico

## 4.1. Antecedentes

### 4.1.1. Evolución de la Computación en la Nube y la Alta Disponibilidad

Con la creciente dependencia de las aplicaciones en línea y la necesidad de ofrecer servicios ininterrumpidos a los usuarios, la alta disponibilidad se ha convertido en un pilar fundamental en el diseño de arquitecturas modernas de TI. La computación en la nube ha facilitado la implementación de infraestructuras resilientes, donde la alta disponibilidad ya no depende de configuraciones de hardware costosas y complicadas, sino de estrategias de replicación y distribución a través de múltiples zonas y regiones geográficas.

Los proveedores de servicios en la nube, como Microsoft Azure, ofrecen soluciones integradas para garantizar que las aplicaciones estén disponibles en todo momento, incluso frente a fallos de hardware, picos de tráfico inesperados o desastres naturales. Las técnicas como la replicación de datos y la conmutación por error entre regiones han permitido que las aplicaciones críticas continúen operando con un tiempo de inactividad mínimo, asegurando la continuidad del negocio y mejorando la experiencia del usuario final [3].Como se visualiza en **Figura 1** se muestra un gráfico que ilustra un servidor en la nube tomada de [4].

**Figura 1.** Computación en la nube

### 4.1.2. Importancia del Tiempo de Escalamiento Automático en la Nube

El tiempo de escalamiento automático se refiere al período que tarda un sistema en ajustar sus recursos de manera dinámica en respuesta a cambios en la demanda de carga. En arquitecturas tradicionales, el escalamiento solía ser un proceso manual y laborioso, lo que podía llevar a tiempos de inactividad o a la degradación del rendimiento durante períodos de alta demanda.

En la actualidad, la computación en la nube permite el escalamiento automático, donde los recursos se ajustan en tiempo real según la necesidad. Sin embargo, el tiempo que tarda el sistema en realizar estos ajustes es crítico. Un tiempo de escalamiento largo puede llevar a una saturación de los recursos disponibles, provocando retrasos en el procesamiento de solicitudes y una mala experiencia de usuario. Por otro lado, un escalamiento demasiado rápido puede resultar en un uso ineficiente de los recursos y un aumento innecesario en los costos operativos [5].

Estudios recientes han señalado que el tiempo de escalamiento puede variar significativamente entre diferentes proveedores de servicios en la nube e incluso entre diferentes configuraciones dentro del mismo proveedor. Factores como la latencia de inicio en frío, la capacidad del sistema para predecir la carga, y la eficiencia de los algoritmos de escalamiento juegan un papel crucial en la optimización del tiempo de respuesta de una aplicación [6].

### 4.1.3. Alta Disponibilidad y su Relación con el Escalamiento Automático

La alta disponibilidad y el escalamiento automático están intrínsecamente relacionados en el diseño de aplicaciones modernas. Un sistema altamente disponible no solo debe estar diseñado para soportar fallos, sino también para escalar de manera eficiente y rápida en respuesta a cambios en la carga de trabajo. Este proceso de escalamiento debe ser lo suficientemente rápido para que el sistema mantenga su capacidad de respuesta, incluso durante picos de tráfico significativos.

Los desafíos de implementar un sistema altamente disponible con un escalamiento automático eficiente incluyen la minimización del tiempo de escalamiento, la reducción del impacto del inicio en frío de las funciones serverless, y la optimización del uso de recursos en la nube para evitar la sobre provisión. Además, es crucial realizar pruebas exhaustivas de carga y estrés para validar la capacidad del sistema de reaccionar adecuadamente bajo diferentes escenarios de carga [7].

### 4.1.4. Estrategias de Optimización del Tiempo de Escalamiento

Optimizar el tiempo de escalamiento requiere un enfoque basado en pruebas y en la comprensión de los patrones de uso de la aplicación. Herramientas como Azure Monitor y Azure Autoscale permiten a los administradores configurar políticas de escalamiento basadas en métricas específicas, como la utilización de CPU o la latencia de las solicitudes. Además, la adopción de arquitecturas serverless ha permitido que las aplicaciones se beneficien de un escalamiento más granular, donde las funciones se activan solo cuando es necesario, mejorando la eficiencia y reduciendo costos [8].

Otro aspecto importante es la configuración adecuada de las zonas de disponibilidad y la replicación de datos. Al distribuir las cargas de trabajo a través de múltiples zonas, se puede asegurar que el sistema mantenga un rendimiento óptimo incluso en caso de fallos en una parte de la infraestructura. Esto no solo mejora la alta disponibilidad, sino que también contribuye a un escalamiento más rápido y eficiente [5].

## 4.2. Tecnologías

### 4.2.1. Computación en la Nube

La computación en la nube ha emergido como una tecnología transformadora, redefiniendo la manera en que las organizaciones desarrollan, despliegan y gestionan aplicaciones. La nube ofrece un modelo de servicio en el cual los recursos computacionales, como almacenamiento, procesamiento y redes, se entregan a través de internet bajo demanda, permitiendo a las empresas escalar sus operaciones de manera flexible y eficiente [9].

Existen tres modelos principales de servicio en la nube:

- **Infraestructura como Servicio (IaaS):** Proporciona recursos básicos de computación como servidores, almacenamiento y redes virtualizadas. Permite a las organizaciones alquilar estas infraestructuras y gestionarlas según sus necesidades [10].
- **Plataforma como Servicio (PaaS):** Ofrece una plataforma completa para el desarrollo y despliegue de aplicaciones, incluyendo herramientas de desarrollo, bases de datos y middleware. Esto facilita a los desarrolladores centrarse en la creación de software sin preocuparse por la gestión subyacente de la infraestructura [11].
- **Software como Servicio (SaaS):** Proporciona aplicaciones listas para usar que se entregan a través de internet, eliminando la necesidad de instalaciones locales y facilitando el acceso a las aplicaciones desde cualquier lugar con conexión a internet [12].

La adopción de la computación en la nube está impulsada por sus múltiples beneficios, como la reducción de costos, la escalabilidad, la flexibilidad y la capacidad de recuperación ante desastres [13]. Las organizaciones pueden escalar sus recursos de forma dinámica, lo que es esencial en entornos donde la demanda de recursos puede variar considerablemente en periodos cortos de tiempo. Como se visualiza en **Figura 2** tomada de [13].

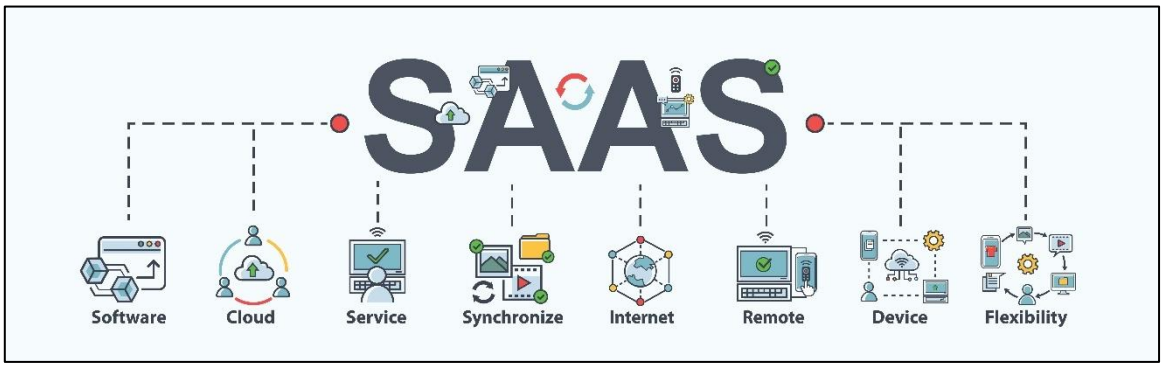



**Figura 2.** Software como Servicio (SaaS)

### 4.2.2. Azure: Plataforma de Computación en la Nube

Microsoft Azure es una plataforma de computación en la nube que ofrece una amplia gama de servicios, incluyendo máquinas virtuales, almacenamiento, bases de datos, redes, y capacidades de inteligencia artificial. Azure permite a las organizaciones construir, desplegar y gestionar aplicaciones a través de una red global de centros de datos, proporcionando tanto escalabilidad como alta disponibilidad [14]

#### Servicios Clave de Azure

Azure proporciona varios servicios diseñados para soportar aplicaciones empresariales críticas:

- **Azure Virtual Machines:** Ofrece instancias de máquinas virtuales configurables que permiten ejecutar aplicaciones y servicios de manera flexible.
- **Azure Functions:** Un servicio serverless que permite ejecutar fragmentos de código en respuesta a eventos sin la necesidad de gestionar la infraestructura subyacente. Es ideal para aplicaciones que necesitan escalar dinámicamente según la carga [15]. Véase la **Figura 3** obtenida de [15].
- **Azure Logic Apps:** Permite la automatización y orquestación de flujos de trabajo que integran servicios en la nube, aplicaciones empresariales y sistemas locales.

- **Azure Kubernetes Service (AKS):** es un servicio de Kubernetes totalmente administrado que simplifica la administración de clústeres y reduce la sobrecarga operativa. Ofrece escalado automático para un escalado de aplicaciones sin problemas, características de seguridad de nivel empresarial e integración con herramientas de desarrollo populares y canalizaciones de CI/CD [16].

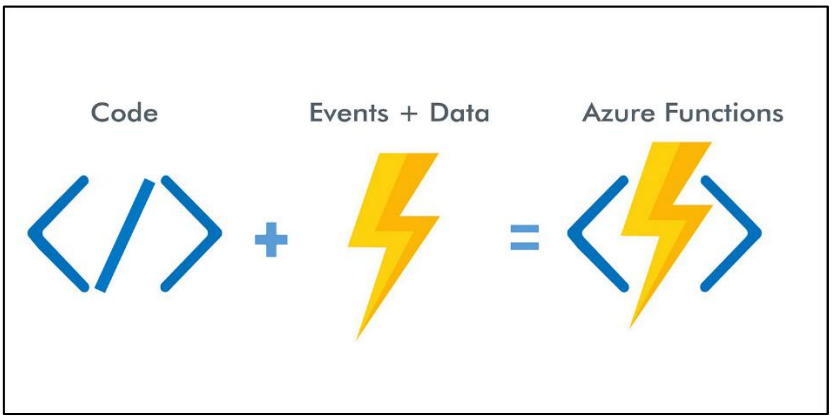



**Figura 3.** Creación de Azure Functions

Azure es conocida por su fiabilidad y por ofrecer herramientas robustas para la gestión de aplicaciones empresariales, permitiendo a las organizaciones mantener un rendimiento consistente y confiable, incluso bajo cargas de trabajo variables.

### 4.2.3. Arquitectura Serverless

La arquitectura serverless es un modelo de computación en la nube en el cual el proveedor gestiona automáticamente la infraestructura necesaria para ejecutar aplicaciones, eliminando la necesidad de que los desarrolladores gestionen servidores y otros componentes de infraestructura. Este enfoque permite que las aplicaciones se escalen automáticamente según la demanda, mejorando la eficiencia y reduciendo costos [16].

**Ventajas de la Arquitectura Serverless**

Algunas de las principales ventajas de la arquitectura serverless incluyen:

- **Escalabilidad automática:** Los recursos se ajustan automáticamente para manejar las variaciones en la carga de trabajo, lo que es crucial para aplicaciones que experimentan picos de demanda [6].
- **Reducción de costos:** Se paga solo por el tiempo de ejecución y los recursos utilizados, lo que puede resultar en un ahorro significativo en comparación con las arquitecturas tradicionales [6].

- **Desarrollo ágil:** Los desarrolladores pueden centrarse en la lógica de negocio sin preocuparse por la administración de servidores [17].

**Desafíos de la Arquitectura Serverless**

A pesar de sus ventajas, la arquitectura serverless también presenta desafíos:

- **Latencia de inicio en frío:** Cuando una función serverless no se ha ejecutado recientemente, puede haber una latencia adicional conocida como "inicio en frío", que puede afectar el rendimiento de aplicaciones sensibles al tiempo [18].
- **Dificultades en la depuración:** La naturaleza distribuida y asíncrona de las aplicaciones serverless puede complicar la depuración y el monitoreo de errores [19].
- **Dependencia del proveedor:** El uso de servicios serverless tiende a crear una fuerte dependencia del proveedor de la nube, lo que puede limitar la flexibilidad a la hora de migrar aplicaciones a otra plataforma [19].

### 4.2.4. Alta Disponibilidad en la Nube

La alta disponibilidad es un requisito fundamental para aplicaciones críticas que deben estar operativas casi todo el tiempo. Esto se refiere a la capacidad de un sistema para continuar funcionando incluso en caso de fallos de hardware, software o red. En la nube, la alta disponibilidad se logra mediante la replicación de datos y la distribución de cargas de trabajo a través de múltiples zonas de disponibilidad y regiones geográficas [14]

**Herramientas de Alta Disponibilidad en Azure**

Azure proporciona varias herramientas y servicios para garantizar la alta disponibilidad de las aplicaciones:

- **Azure Load Balancer:** Distribuye el tráfico de red entrante entre varias máquinas virtuales, garantizando la disponibilidad y la resiliencia del servicio [13].
- **Azure Traffic Manager:** Un servicio de DNS basado en la nube que permite distribuir el tráfico a servicios específicos de Azure en diferentes ubicaciones globales, lo que mejora la disponibilidad y la capacidad de respuesta [20].
- **Azure Site Recovery:** Ofrece una solución integral para la recuperación ante desastres, replicando cargas de trabajo críticas a una ubicación secundaria para garantizar la continuidad del negocio en caso de fallos

### 4.2.5. Tiempo de Escalamiento Automático

El tiempo de escalamiento automático es un factor crucial en las arquitecturas serverless, ya que determina cuánto tiempo tarda un sistema en ajustar sus recursos en respuesta a un aumento en la carga de trabajo. Este tiempo de respuesta es vital para mantener la calidad del servicio y evitar interrupciones, especialmente en aplicaciones que requieren alta disponibilidad [21].

Investigaciones recientes han mostrado que el tiempo de escalamiento puede variar significativamente dependiendo de la configuración y el proveedor de servicios en la nube, lo que subraya la importancia de optimizar estos tiempos para asegurar que las aplicaciones sean capaces de manejar picos de demanda sin degradación en el rendimiento [3]

**Importancia del Tiempo de Escalamiento Automático**

El tiempo de escalamiento automático es un factor crucial en las arquitecturas serverless, ya que define cuánto tiempo tarda un sistema en reaccionar a un aumento en la carga de trabajo. Este tiempo de respuesta es esencial para mantener la calidad del servicio, especialmente en aplicaciones de misión crítica, como los sistemas de gestión de reservas [5].

Optimizar el tiempo de escalamiento es fundamental para garantizar que las aplicaciones permanezcan disponibles y receptivas, minimizando el impacto de los picos de demanda en la experiencia del usuario final.

### 4.2.6. Lenguajes de Programación

En el contexto de arquitecturas **serverless**, la selección de la tecnología adecuada es crucial para garantizar un rendimiento óptimo, compatibilidad con el entorno cloud y escalabilidad. A continuación, se comparan las tecnologías más utilizadas para la implementación de funciones serverless, incluyendo **Node.js**, **Python** y **Go**, destacando las características clave que influyen en su elección.

**Tabla 1.** Comparación entre diferentes tecnologías de Implementación

| Característica | Node | Python | Go | Spring Boot (Java) |
|---|---|---|---|---|
| Velocidad de Inicio en Frío | Rápida | Moderada | Muy rápida | Lenta |
| Ecosistema de Librerías | Amplio y diverso | Amplio y maduro | Menor, pero eficiente | Muy amplio |
| Compatibilidad con Azure | Alta | Alta | Alta | Alta |
| Manejo de Concurrencia | Excelente (event-driven) | Bueno (threads) | Excelente (goroutines) | Moderado (threads) |
| Facilidad de Uso | Accesible (JavaScript) | Muy accesible (Python) | Moderada | Compleja (verbose) |
| Adaptación Serverless | Natural | Natural | Natural | Requiere adaptación |

**Velocidad de Inicio en Frío**

Node.js y Go destacan por sus tiempos de inicio en frío rápidos, gracias a su naturaleza ligera. En contraste, **Spring Boot** sufre de tiempos de inicio en frío más largos debido a la carga inicial del framework y la **Java Virtual Machine (JVM)**. Este tiempo adicional puede ser crítico en arquitecturas serverless donde la latencia inicial debe mantenerse al mínimo [22].

**Manejo de Concurrencia**

Node.js y Go sobresalen en el manejo de concurrencia gracias a sus modelos basados en eventos y **goroutines**, respectivamente. **Spring Boot**, aunque robusto, se basa en un modelo de hilos tradicional, lo que puede consumir más recursos y resultar menos eficiente en entornos serverless donde la escalabilidad horizontal es clave.

**Ecosistema de Librerías y Herramientas**

Spring Boot tiene un ecosistema robusto y maduro, especialmente para aplicaciones empresariales. Sin embargo, para arquitecturas serverless, este ecosistema puede ser más pesado en comparación con Node.js, que cuenta con una integración más directa con entornos serverless y un conjunto de herramientas livianas a través de **npm**.

### Adaptación para Arquitecturas Serverless

Mientras que Node.js, Python y Go tienen una integración natural con Azure Functions, **Spring Boot** requiere adaptaciones adicionales para optimizar su rendimiento en entornos serverless. Esto incluye el uso de estrategias como **Spring Cloud Function** para hacer que las aplicaciones Spring sean más adecuadas para el despliegue serverless [23].

### Facilidad de Uso y Complejidad

Spring Boot es más complejo y verboso que Node.js y Python, especialmente para desarrolladores que no están familiarizados con el ecosistema Java. Esto puede aumentar los tiempos de desarrollo inicial en proyectos como el sistema de reservas.

## 4.3. Trabajos Relacionados

**Tabla 2.** Trabajos relacionados

| Trabajo | Resumen | Autor |
|---|---|---|
| Evaluación del Escalamiento Automático en la Computación en la Nube | Este trabajo analiza el comportamiento del escalamiento automático en diferentes plataformas de computación en la nube, compa eficiencia y los tiempos de respuesta. Los autores identifican los en la configuración de políticas de escalamiento que puedan adaptabilidad y variaciones rápidas en la carga de trabajo, destacando el impacto de la optimización personalizada para mejorar la disponibilidad sistema [21]. | Aaron Brewer |
| Arquitecturas Serverless y su Eficiencia en el Escalamiento Automático | Este artículo explora la relación entre las arquitecturas server eficiencia del escalamiento automático. Se analizan casos de donde las funciones serverless son utilizadas para manejar demanda, mostrando cómo el inicio en frío y la configuración de escalamiento afectan el rendimiento general del sistema [8]. | Rajesh Ranjan |
| Optimización del Escalamiento Automático en Azure | El documento presenta un análisis detallado de la estrategia de optimización del tiempo de escalamiento en Azure, con un enfoque utilización de herramientas como Azure Monitor y Azure Au Los autores ofrecen recomendaciones prácticas para minimizar tiempo de escalamiento y mejorar la eficiencia operativa en entorno disponibilidad [10]. | Zhang Qi |
| Implementación de una arquitectura escalable basada en Google Cloud Platform para mejorar la disponibilidad y escalabilidad de información de la | Este trabajo presenta una propuesta de arquitectura escalable utilizando Google Cloud Platform para mejorar la disponibilidad y escalabilidad de la información en la empresa SmartBrands S.A.C. Los resultados muestran un incremento en la disponibilidad del ERP en un 1.5% y una mejora del 2.19% respecto al servidor on-premise, analizando los primeros meses del año 2020 [24]. | Ricardo Miguel Llontop García |

| | | |
|---|---|---|
| empresa SmartBrands | | |
| Implementación de una arquitectura de computación en la nube (Cloud Computing) diseñada para escalabilidad automática y alta disponibilidad basado en la plataforma de Amazon Web Services (AWS) en la Universidad de Lambayeque | Esta investigación implementa una arquitectura de computación en la nube diseñada para escalabilidad automática y alta disponibilidad en la Universidad de Lambayeque, utilizando Amazon Web Services (AWS). Los resultados indican que la aplicación soporta más de 400 usuarios concurrentes, optimizando el uso de recursos y mejorando la eficiencia [25]. | Martín David Llauce Santos |
| Aplicación de tecnologías y arquitecturas serverless para el desarrollo de soluciones IoT | Este proyecto investiga el concepto de arquitectura serverless o FaaS (Functions as a Service) para soluciones IoT, proponiendo la implementación de soluciones escalables, flexibles, estables y con costes bajos en entornos de Internet de las Cosas [26]. | Eduardo Sanfrutos |
| Análisis comparativo de las arquitecturas serverless, que los diferentes proveedores, ofrecen en la nube orientado a la creación de una aplicación web | El proyecto pretende dar a conocer un poco más esta arquitectura que muchas empresas ya están optando por utilizarla y muchas otras aún no conocen del gran potencial que tiene trabajar bajo la arquitectura serverless y los grandes beneficios que conlleva [27]. | José Nicolás Mayanquer Rosero |

## 5. Metodología

La metodología DevOps fue seleccionada para garantizar una integración continua y despliegue eficiente de la arquitectura serverless. Este enfoque combina herramientas, procesos y colaboración para optimizar el tiempo de desarrollo y asegurar la calidad del sistema.

La metodología que se busca abordar en este proyecto se describe a continuación:

Para resolver el **Objetivo 1**: *"Implementar la arquitectura Serverless con alta disponibilidad para la Gestión de reservas usando Azure mediante la metodología DevOps"*, se desarrolló basada en

**Configuración de Servicios Serverless**:

- Configuración y despliegue de **Azure Functions** para manejar las operaciones críticas del sistema.
- Configuración de **Azure App Service** como complemento para servicios que no sean estrictamente serverless.

**Integración del Sistema de Gestión de Reservas**:

- Adaptación de los módulos **AuthCore** y **BookingCore** para integrarse con servicios de escalamiento automático en Azure.
- Configuración de bases de datos en **Azure SQL** con conexiones seguras mediante **Azure Key Vault**.

**Documentación del Proceso**:

- Documentación detallada de la configuración y arquitectura, incluyendo diagramas y scripts de despliegue.

Para resolver el **Objetivo 2:** *"Determinar el tiempo de escalamiento usando pruebas de carga y estrés mediante pruebas de hipótesis"*, se implementó la metodología:

**1. Configuración del Entorno de Pruebas**

1. **Preparación de la Infraestructura:**
    - Configuración de **Azure Functions** con el plan de consumo dinámico habilitado para escalamiento automático.
    - Definición de métricas clave de monitoreo, como latencia promedio, uso de CPU y memoria.
2. **Herramientas de Pruebas:**
    - Instalación y configuración de **Apache JMeter** para la simulación de cargas.
    - Integración de **Azure Monitor** y **Application Insights** para registrar métricas en tiempo real.

**2. Diseño y Ejecución de Pruebas**

1. **Conjunto de Pruebas de Carga:**
    - Diseño de escenarios de carga que incrementen progresivamente el número de solicitudes concurrentes desde valores bajos hasta valores críticos.
    - Simulación de solicitudes HTTP que interactúan con las funciones serverless configuradas.
2. **Pruebas de Estrés:**
    - Creación de escenarios extremos que excedan la capacidad teórica del sistema para identificar límites de escalabilidad.
    - Generación de picos de demanda súbitos para evaluar el tiempo de respuesta inicial del escalamiento automático.
3. **Ejecución de Pruebas:**
    - Ejecución de múltiples rondas de pruebas para asegurar consistencia en los resultados.
    - Registro de métricas como el tiempo promedio de escalamiento y el tiempo de respuesta bajo carga.

### 3. Análisis de Resultados

1. **Procesamiento de Métricas:**
    - Recopilación de datos desde **JMeter**, **Azure Monitor**, y **Application Insights**.
    - Organización de los datos en tablas comparativas para análisis estadístico.
2. **Pruebas de Hipótesis:**
    - Formulación de hipótesis para validar la efectividad del escalamiento automático:
        - Hipótesis nula ($H_0$): El tiempo de escalamiento automático no cumple con el umbral aceptable definido.
        - Hipótesis alternativa ($H_1$): El tiempo de escalamiento automático cumple con el umbral aceptable.
    - Aplicación de pruebas estadísticas (como t-test) para analizar los tiempos de escalamiento bajo diferentes condiciones de carga.
3. **Informe Comparativo:**
    - Comparación de resultados obtenidos en escenarios de carga y estrés.
    - Identificación de patrones o problemas recurrentes en el tiempo de respuesta y escalamiento.

### Procesamiento y Análisis de Datos

1. **Recolección de Datos:**
    - Datos recopilados en tiempo real durante las pruebas, como latencia promedio, tiempo de escalamiento y uso de recursos.
2. **Visualización y Comparación:**
    - Generación de gráficos de dispersión y líneas de tiempo para comparar el desempeño del sistema en distintos niveles de carga.
3. **Conclusión Estadística:**
    - Validación de los resultados mediante pruebas de hipótesis, destacando si el sistema cumple con los criterios establecidos para el tiempo de escalamiento.

## 6. Resultados

La implementación de la arquitectura serverless con alta disponibilidad para la gestión de reservas permitió alcanzar los objetivos propuestos, garantizando un sistema escalable y resiliente mediante el uso de Azure Functions. Los resultados obtenidos se presentan a continuación:

### 6.1. Implementar la arquitectura Serverless con alta disponibilidad para la Gestión de reservas usando Azure mediante la metodología DevOps

El sistema de Gestión de Reservas el cual fue objeto de estudio se construyó tomando en cuenta las necesidades descritas en las siguientes historias de usuario.

Un usuario es la persona que interactúa con el sistema y hace la reserva de un asiento para un determinado evento. En la **Tabla 3** se presenta la historia de usuario correspondiente al proceso de registro de un nuevo usuario.

**Tabla 3.** Historia de Usuario Registro de usuario

| **Nombre** | **Registro de usuario** |
|---|---|
| **Descripción** | Como un visitante,<br>quiero registrarme con mis datos personales,<br>para poder acceder al sistema y realizar reservas. |
| **Criterios de aceptación** | |
| 1 | El sistema debe validar que el correo electrónico sea único. |
| 2 | El sistema debe cifrar la contraseña antes de almacenarla. |
| 3 | El usuario debe recibir un mensaje de éxito al registrarse. |

El inicio de sesión permite que un usuario pueda acceder al sistema con credenciales previamente creadas. En la siguiente **Tabla 4** se presenta la información para el proceso de iniciar sesión.

**Tabla 4.** Historia de Usuario Inicio de sesión

| **Nombre** | **Inicio de sesión** |
|---|---|
| **Descripción** | Como un usuario registrado,<br>quiero iniciar sesión con mi correo y contraseña,<br>para acceder a mis reservas y gestionar mi cuenta. |
| **Criterios de aceptación** | |
| 1 | El sistema debe validar las credenciales. |
| 2 | Si las credenciales son incorrectas, debe mostrar un mensaje genérico: "Credenciales no válidas". |
| 3 | Debe generar y devolver un token JWT válido. |

Cada usuario tiene un rol asignado, los roles se pueden administrar según las necesidades del administrador del sistema, en la siguiente **Tabla 5** se muestra más información.

**Tabla 5.** Historia de Usuario Gestión de roles

| **Nombre** | **Gestión de roles** |
|---|---|
| **Descripción** | Como un administrador,<br>quiero asignar roles a los usuarios,<br>para limitar o permitir accesos según las necesidades. |
| **Criterios de aceptación** | |
| 1 | Los roles deben ser gestionados únicamente por usuarios con permisos administrativos. |
| 2 | Cada usuario puede tener un único rol asignado. |

Los eventos son entidades en las cuales se pueden hacer reservas, como puede ser un hotel, concierto. Un evento contiene Tipos de asientos y a su vez asientos. Véase la **Tabla 6** para más información.

**Tabla 6.** Historia de Usuario Gestión de eventos

| **Nombre** | **Gestión de eventos** |
|---|---|
| **Descripción** | Como un administrador,<br>Quiero gestionar eventos<br>para crear eventos con información detallada,<br>modificar y eliminar eventos<br>para permitir la reserva de asientos. |
| **Criterios de aceptación** | |
| 1 | El evento debe incluir nombre, descripción, dirección y capacidad total. |
| 2 | Debe permitir agregar coordenadas de ubicación (latitud y longitud). |
| 3 | Se debe poder modificar y poder eliminar un evento |

Cada evento contiene uno o varios tipos de asientos, estos tipos hace referencia a la división que existe ya que cada asiento tiene un precio diferente dependiendo la localización.

**Tabla 7.** Historia de Usuario Gestión de tipos de asientos

| **Nombre** | **Gestión de tipos de asientos** |
|---|---|
| **Descripción** | Como un administrador,<br>quiero definir tipos de asientos (VIP, General, etc.),<br>para asociarlos con eventos y establecer precios. |
| **Criterios de aceptación** | |
| 1 | Los tipos de asientos deben tener un precio asociado. |
| 2 | Cada tipo de asiento debe estar vinculado a un evento específico. |
| 3 | Cada tipo de asiento tiene muchos asientos asociados |

La reserva consiste en separar un asiento disponible para un usuario del sistema. Un usuario puede hacer muchas reservas siempre y cuando estes disponibles. Después que se haya reservado un asiento su estado cambia. Véase la siguiente **Tabla 8** para un mayor detalle.

**Tabla 8.** Historia de Usuario Creación de reservas

| **Nombre** | **Creación de reservas** |
|---|---|
| **Descripción** | Como un usuario autenticado,<br>quiero reservar asientos para un evento,<br>para asegurar mi lugar en el evento. |
| **Criterios de aceptación** | |
| 1 | El sistema debe validar que los asientos seleccionados estén disponibles. |
| 2 | El sistema debe calcular el costo total en función de los precios de los asientos seleccionados. |
| 3 | El sistema debe cambiar el estado de los asientos reservados a "reservado". |

En el caso que se haya hecho una reserva y se requiera cancelar se debe cambiar el estado del asiento para que quede disponible para demás usuarios además de eliminar la reserva creada. Puede verse la **Tabla 9** para un mayor detalle.

**Tabla 9.** Historia de Usuario Cancelación de reservas

| **Nombre** | **Cancelación de reservas** |
|---|---|
| **Descripción** | Como un usuario autenticado,<br>quiero cancelar una reserva,<br>para liberar los asientos y actualizar mi estado. |
| **Criterios de aceptación** | |
| 1 | El sistema debe cambiar el estado de los asientos reservados a "disponible". |
| 2 | El sistema debe registrar la cancelación en la base de datos. |

Los usuarios podrán listar todas las reservas que hagan hecho, se les deberá mostrar la información de la reserva, del tipo de asiento asociado y del evento al cual pertenece. Véase la

**Tabla 10** para un mayor detalle.

**Tabla 10.** Historia de Usuario Listado de reservas

| **Nombre** | **Listado de reservas** |
|---|---|
| **Descripción** | Como un usuario autenticado,<br>quiero visualizar mis reservas paginadas,<br>para acceder fácilmente a la información de mis eventos. |
| **Criterios de aceptación** | |
| 1 | El sistema debe mostrar solo las reservas activas y asociadas al usuario. |
| 2 | Debe incluir la información básica de la reserva y el costo total. |
| 3 | Se debe mostrar el tipo de asiento y evento asociado a los asientos de la reserva |

Se debe mostrar los asientos disponibles para un determinado evento en un rango de fechas con el fin que el usuario pueda buscar y seleccionar la mejor opción para realizar una reserva. Puede verse el detalle completo en la siguiente **Tabla 11**.

**Tabla 11.** Historia de Usuario Visualización de asientos disponibles

| **Nombre** | **Visualización de asientos disponibles** |
|---|---|
| **Descripción** | Como un usuario autenticado,<br>quiero ver los asientos disponibles para un evento,<br>para elegir el lugar que prefiero. |
| **Criterios de aceptación** | |
| 1 | Los asientos deben mostrarse según su tipo y estado. |
| 2 | Los asientos no disponibles deben estar claramente marcados. |

Cada asiento debe tener diferentes estados que permitan conocer su disponibilidad para un determinado evento. Véase la siguiente **Tabla 12** para más información.

**Tabla 12.** Historia de Usuario Gestión de estados de asientos

| **Nombre** | **Gestión de estados de asientos** |
|---|---|
| **Descripción** | Como un administrador,<br>quiero actualizar el estado de los asientos,<br>para reflejar cambios en la disponibilidad. |
| **Criterios de aceptación** | |
| 1 | El sistema debe permitir cambiar el estado a "disponible" o "reservado". |
| 2 | Debe mantener un registro de cambios en los estados. |

Dentro de Azure, se creó un Resource Group denominado maestría. Se creo el recurso de aplicación Función App, lo que se denomina una función Serverless

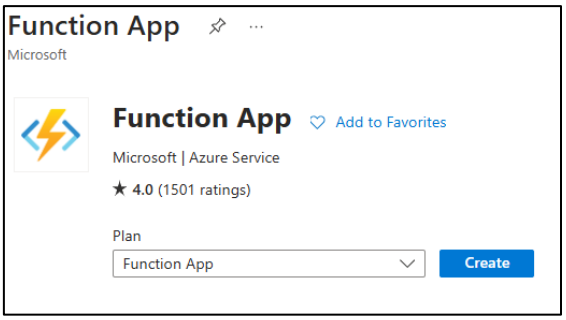



**Figura 4.** Function App en Azure

El plan que se selecciono fue **Consumption** dado las características del máximo de escalamiento y es mejor en relación costo beneficio.

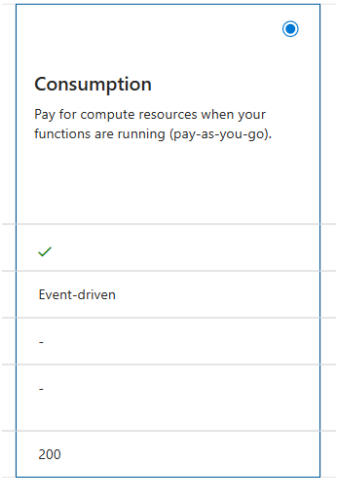



**Figura 5.** Plan Consumption para crear Function App

Se agrega la configuración de necesaria, como la suscripción, nombre, región, sistema operativo.

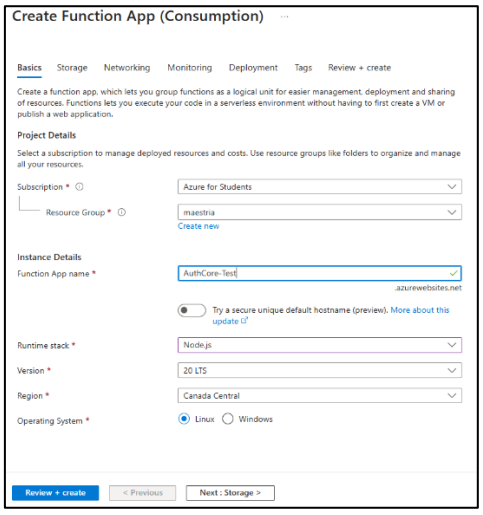



**Figura 6.** Configuración creación Function App

Se selecciono la configuración de almacenamiento normal, se recomienda agregar valores según la necesidad del proyecto.

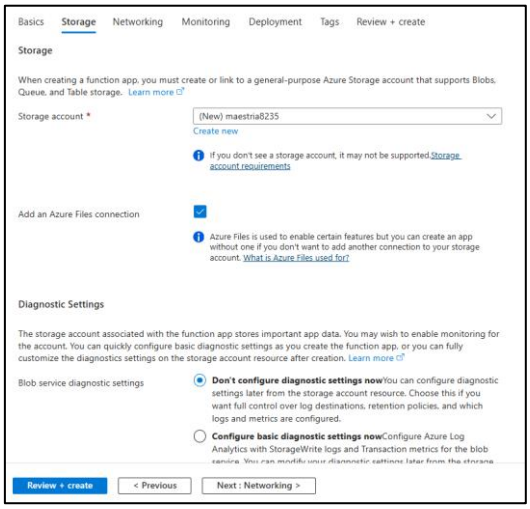



**Figura 7.** Configuración almacenamiento Function App

Luego de haber agregado todas las configuraciones necesarias se debe ir a la sección de “Review + Create” y aplastar el botón de “Create”

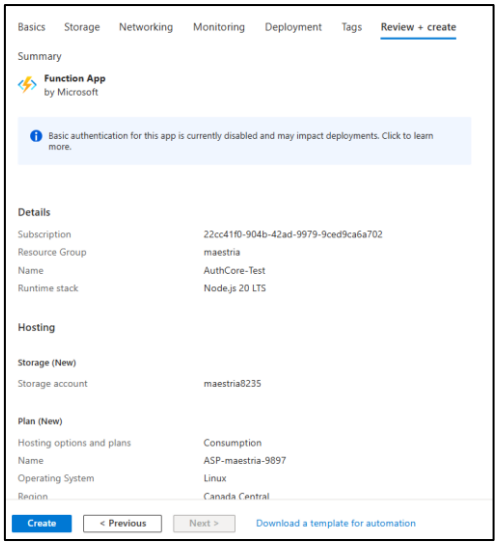



**Figura 8.** Sección de creación de Function App

La aplicación se creó de manera exitosa y muestra todas las opciones configuradas

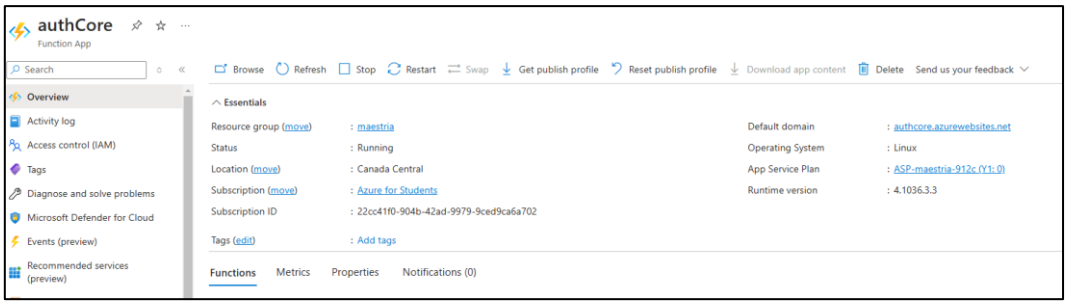



**Figura 9.** Function App creado de manera exitosa

La aplicación serverless (sin servidor) está lista para usarse, hace falta asignar la lógica de la aplicación.

### Arquitectura de la integración en Azure

Se crearon 2 servicios principales los cuales contiene toda la lógica de la Gestión de Reservas.

- **AuthCore**: Gestiona usuarios y roles, validando la autenticidad y autorización para acceder a las funciones relacionadas con las reservas.
- **BookingCore**: Maneja eventos, asientos y reservas, garantizando la consistencia de los datos incluso bajo alta concurrencia.

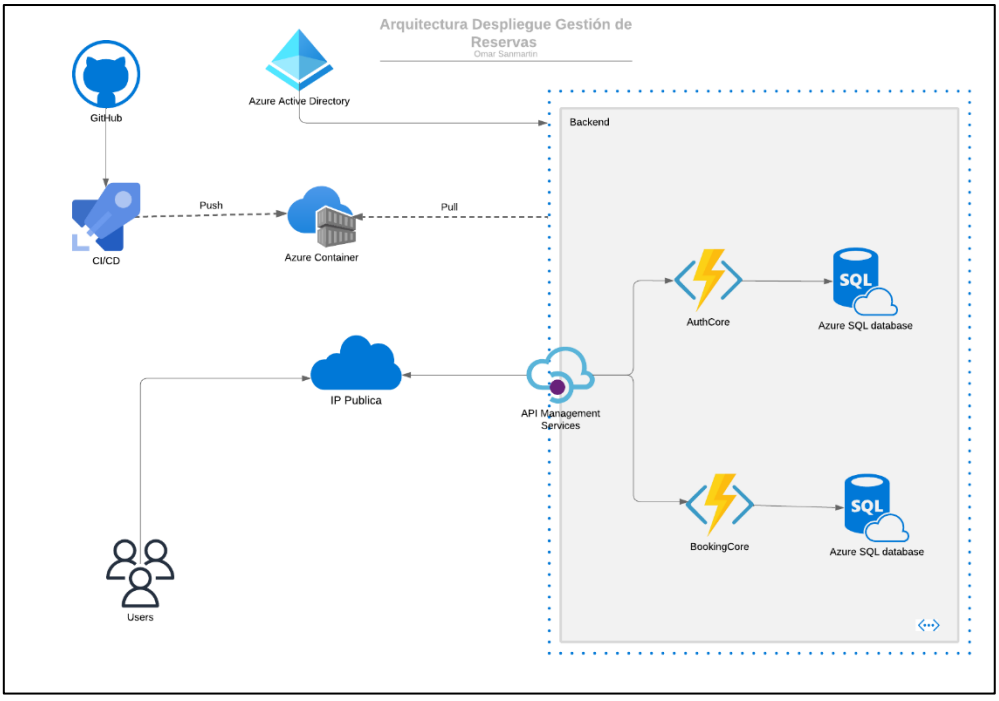



**Figura 10.** Arquitectura de Despliegue de Gestión de Reservas

Para la construcción del sistema de Gestión de Reservas se usó Node.js debido a que es la mejor opción para el sistema de gestión de reservas basado en Azure Functions debido a:

1. Tiempos de inicio en frío competitivos, esenciales para arquitecturas serverless.
2. Modelo de concurrencia eficiente, adecuado para manejar solicitudes simultáneas en un sistema crítico.
3. Integración natural con Azure Functions, lo que facilita su despliegue y mantenimiento.

Como IDE de desarrollo se usó WebStorm de JetBrain, sin embargo, puede ser reemplazado por Visual Studio.

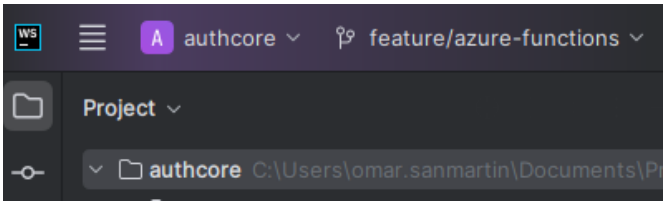



**Figura 11.** IDE de desarrollo

Se construyo el código para los servicios de AuthCore y Bookincore. Las funcionalidades desarrolladas constituyen el Backend de la aplicación con toda la lógica de negoció mencionada en las historias de usuario de Gestión de Reservas.

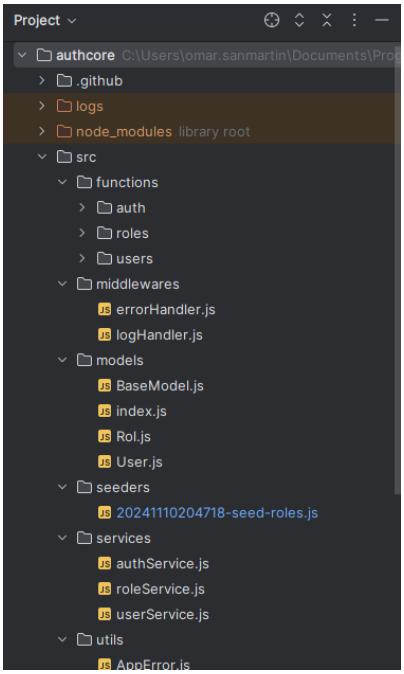



**Figura 12.** Estrcutura de Proyecto authCore

Con Docker se creó la base de datos para los dos servicios, además usar a pgadmin como cliente de base de datos.


**Figura 13.** Base de datos ejecutada en Docker

De manera local se hicieron pruebas de los dos servicios (AuthCore y Bookincore) de forma exhaustiva para asegurar que se cumpla con el correcto funcionamiento. Para probar la aplicación serverless (Sin servidor) se hizo uso de la librería Azure Functions Core Tools, esta herramienta permite desarrollar, ejecutar y depurar funciones localmente antes de desplegarlas en Azure.


**Figura 14.** Libreria Azure Functions Core Tools para desarrollo

Para iniciar la aplicación localmente se hace uso del comando func start


**Figura 15.** Inicio de aplicación de manera local

Después de haber iniciado la aplicación se pueden hacer todas las pruebas de los servicios serverless (Sin servidor)


**Figura 16.** Function App AuthCore ejecutandose

Para verificar el correcto funcionamiento de cada endpoint se usó Postman, donde se crearon 2 colecciones, una para cada servicio, se registraron todos los endpoints disponibles para validar su funcionamiento y verificar si cumple con la lógica propuesta de la Gestión de Reservas.

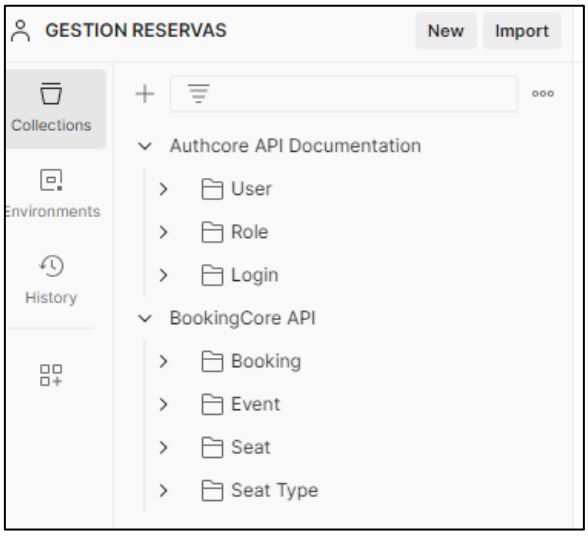
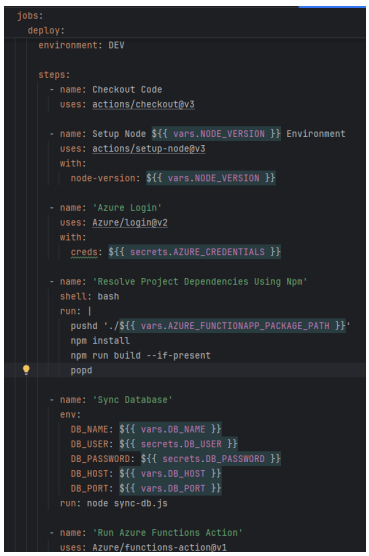



**Figura 17.** Colección de Endpoints en Postman

**Pipeline de Despliegue**

Se creo un pipeline de despliegue que se ejecuta en Github Actions con el fin de desplegar funcionalidades a la nube de manera rápida y continua.

El pipeline automatiza el despliegue de un proyecto Node.js en Azure Function App. Incluye pasos para clonar el repositorio, configurar el entorno de Node.js, instalar dependencias, sincronizar la base de datos, desplegar el proyecto en Azure utilizando un perfil de publicación, y establecer variables de entorno necesarias en la Function App. Se ejecuta automáticamente al hacer un push en la rama master y utiliza credenciales y configuraciones almacenadas de forma segura.



```yaml
jobs:
  deploy:
    environment: DEV

    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Setup Node ${{ vars.NODE_VERSION }} Environment
        uses: actions/setup-node@v3
        with:
          node-version: ${{ vars.NODE_VERSION }}

      - name: 'Azure Login'
        uses: Azure/login@v2
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}

      - name: 'Resolve Project Dependencies Using Npm'
        shell: bash
        run: |
          pushd './${{ vars.AZURE_FUNCTIONAPP_PACKAGE_PATH }}'
          npm install
          npm run build --if-present
          popd

      - name: 'Sync Database'
        env:
          DB_NAME: ${{ vars.DB_NAME }}
          DB_USER: ${{ secrets.DB_USER }}
          DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
          DB_HOST: ${{ vars.DB_HOST }}
          DB_PORT: ${{ vars.DB_PORT }}
        run: node sync-db.js

      - name: 'Run Azure Functions Action'
        uses: Azure/functions-action@v1
```

**Figura 18.** Pipeline Github Actions de despliegue a Azure

Al hacer push en la rama master del repositorio de cada servicio se ejecutará el pipeline y se desplegará la aplicación.

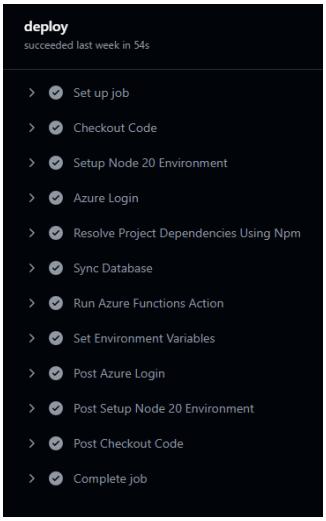



**Figura 19.** Ejecución exitosa en Github Actions

En Azure dentro de la Function App creada se pueden ver todos los Endpoints publicados que pueden usarse.

**authCore**

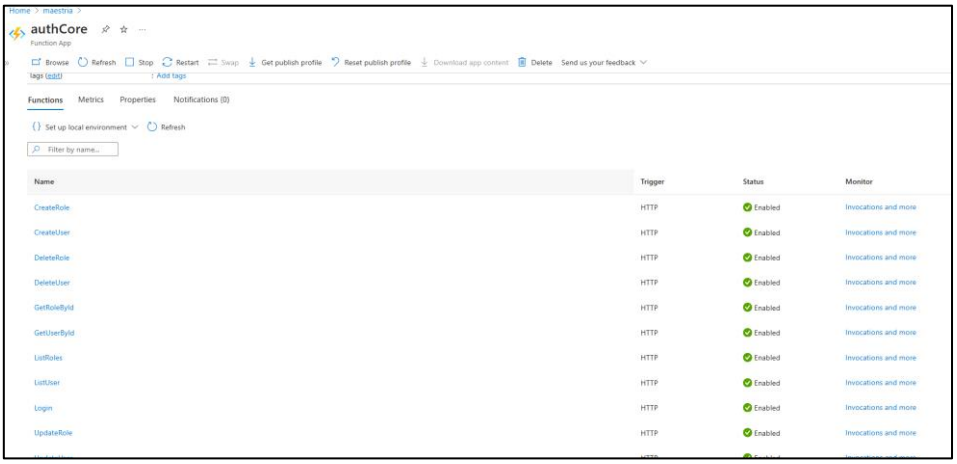



**Figura 20.** Function App authCore en Azure

**bookinCoreApi**

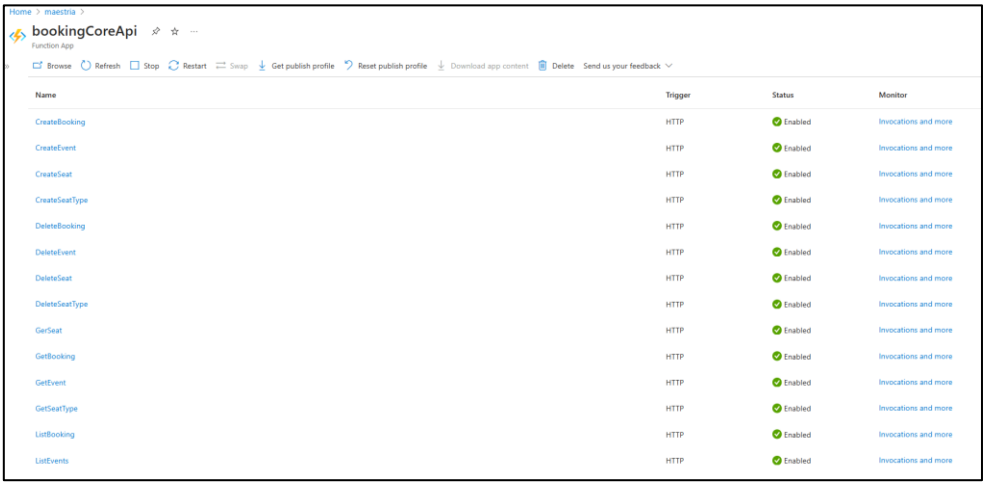



**Figura 21.** Function App bookinCore en Azure

Para obtener información de la implementación hecha en el código véase ¡Error! No se e ncuentra el origen de la referencia.

Para la base de datos por motivos de prueba y de versatilidad se usó Azure Database for PostgreSQL - Flexible Serve, ya que es una base de datos completamente administrada que ofrece flexibilidad y control sobre la configuración, el rendimiento y la disponibilidad.

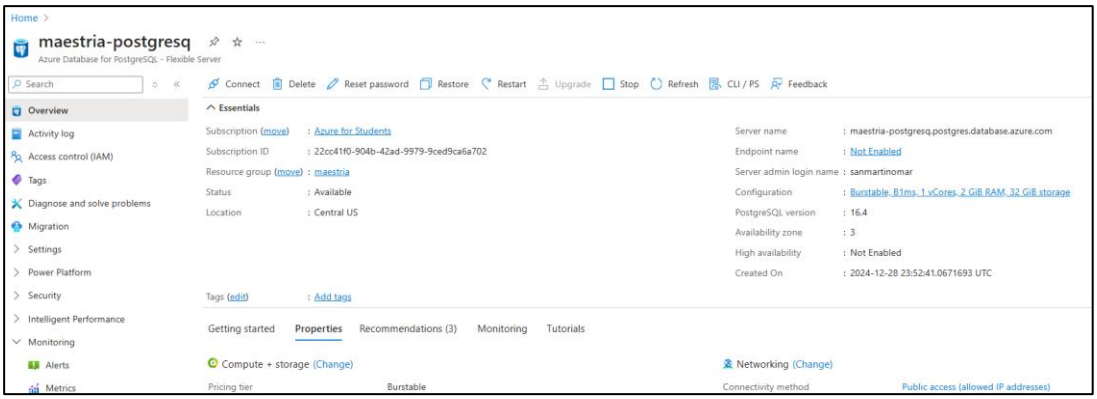



**Figura 22.** Azure Database for PostgreSQL - Flexible Serve

Se configuro Azure API Management como un Gateway para que el punto de entrada sea un endpoint de ruta común.

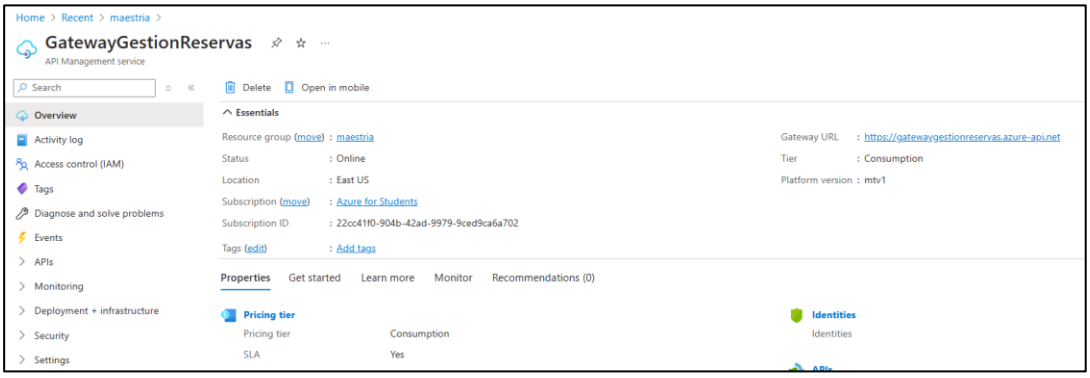



**Figura 23.** API Management Service

Se puede agregar las rutas o aplicaciones (Function App) a las cuales se redireccionará un recurso.


**Figura 24.** Configuración servicio authCore en el Gateway


**Figura 25.** Configuración servicio bookingCore en el Gateway

Desde API Management Service (Gateway) se pudo configurar los permisos, rutas necesarias en cada endpoint de cada servicio.

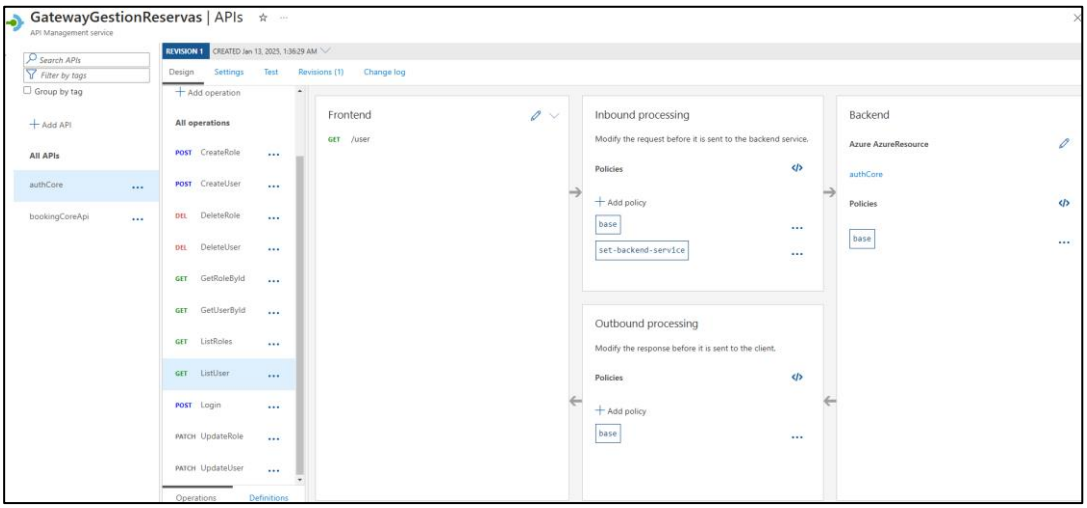



**Figura 26.** Configuración de endpoints desde el Gateway

Para un sistema de gestión de reservas integrado con servicios serverless, usar Azure Database for PostgreSQL - Flexible Server tiene múltiples beneficios como integración Nativa con Azure Functions, Escalabilidad para Demandas Variables, Mantenimiento y Seguridad Simplificados, Costo-Eficiencia.

La configuración adicional que se hizo fue habilitar el rango de direcciones IP para que se haga un update de las tablas a través del pipeline, se tuvo que habilitar la ip publica que usa Github Actions para que pase por el Firewall.

De la misma manera se habilito para que cualquier aplicación dentro de Azure incluyendo las Function App (Sin servidor) como lo son AuthCore y BookingCore tengan acceso. El

Firewall es una herramienta que nos deniega conexiones y debemos hacer configuraciones adicionales, pero es importante para la seguridad de la información al tratarse de una base de datos donde la información es crítica.

Al final se logró la integración del sistema de gestión de reservas es decir los servicios serverless de escalamiento automático de Azure AuthCore y BookingCore con la base de datos Azure Database for PostgreSQL - Flexible Server.

**Documentación Detallada del Proceso de Implementación y configuración**

Se desarrolló una guía técnica que documenta de forma estructurada el proceso de implementación y configuración del sistema:

- **Pasos para configurar y desplegar las funciones en Azure Functions**, incluyendo la creación de recursos en Azure Portal.
- **Diagramas de arquitectura** que muestran la interacción entre los módulos AuthCore y BookingCore, y los servicios de Azure.
- Ejemplos de código para funciones críticas como la creación de usuarios y reservas.

Manual técnico completo replicable para futuras implementaciones o expansiones, véase ¡Error! No se encuentra el origen de la referencia.

El repositorio de los servicios se encuentra en las siguientes direcciones:

https://github.com/omarAlexis1999/authCore.git

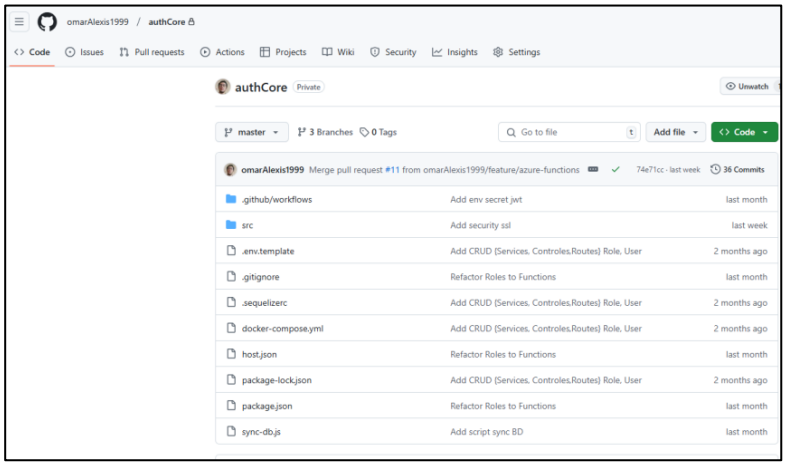



**Figura 27.** Repositorio servicio AuthCore

https://github.com/omarAlexis1999/bookingCore.git

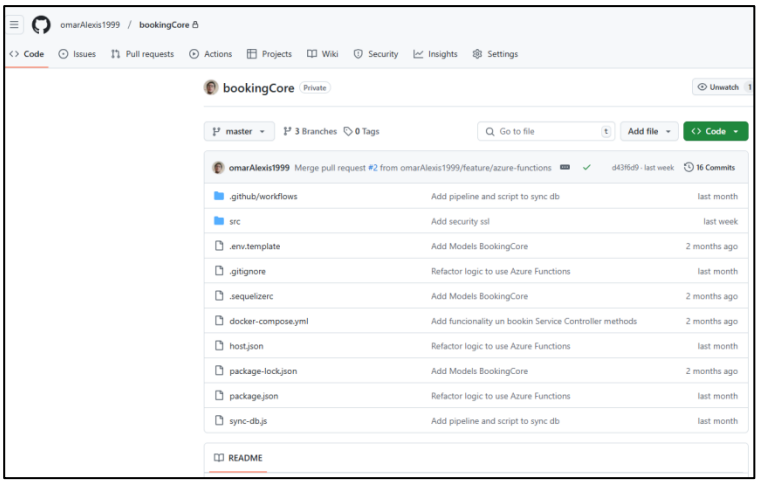



**Figura 28.** Repositorio servicio BookingCore

Tanto el servicio de AuthCore y BookinCore fueron subidos a la nube y expuestos en una URL pública https://gatewaygestionreservas.azure-api.net/

Para obtener más información de URLs expuestas y rutas para cada servicio Véase ¡Error! No se encuentra el origen de la referencia.

## 6.2. Determinar el tiempo de escalamiento usando pruebas de carga y estrés mediante pruebas de hipótesis

### Conjunto de pruebas de carga y estrés

Para validar el funcionamiento del escalamiento se usaron pruebas de:

- **Carga:** Validar el tiempo de respuesta y la estabilidad del sistema bajo un número progresivo de usuarios concurrentes.
- **Estrés:** Identificar el punto de saturación del sistema al simular picos extremos de demanda.

Los servicios para evaluar son:

- **AuthCore:** Encargado de la autenticación y gestión de usuarios.
- **BookingCore:** Encargado de la gestión de eventos, asientos y reservas.

Para realizar las pruebas se hizo uso de la herramienta Jmeter la cual es fácil de usar y muy robusta en cuanto a funcionalidades. A continuación, se detalla todos los recursos usados.

- **Herramientas:**
    - Apache JMeter para la generación de cargas y simulación de usuarios concurrentes.
    - Azure Monitor y Application Insights para la recolección de métricas como tiempo de respuesta, tasa de éxito y número de instancias activas.
- **Recursos:**
    - Infraestructura: Azure Functions (Plan de Consumo).
    - API Management service
    - Base de datos: Azure Database for PostgreSQL - Flexible

El objetivo de las pruebas es analizar el comportamiento del sistema bajo diferentes niveles de carga y estrés, midiendo:

1. **Tiempo de respuesta:** La rapidez con la que los servicios procesan solicitudes.
2. **Tiempo de escalamiento:** El tiempo necesario para que Azure Functions cree nuevas instancias durante picos de carga.
3. **Estabilidad:** La capacidad del sistema para mantener la operatividad sin errores durante escenarios de alta demanda.

Los escenarios que se plantearon son los siguientes:

1. **Pruebas de carga progresiva.**

   Objetivo: Evaluar el tiempo de respuesta y la tasa de éxito al incrementar progresivamente las solicitudes.

   Métrica Clave: Tiempo promedio de respuesta por nivel de carga.

   Niveles de Carga:

   - 200 solicitudes concurrentes.
   - 500 solicitudes concurrentes
   - 500 solicitudes concurrentes

2. **Pruebas de carga sostenida.**

   Objetivo: Verificar la estabilidad del sistema durante periodos prolongados de carga constante.

   Duración: 15 minutos por nivel de carga.

   Niveles de Carga:

   - 500 solicitudes concurrentes.
   - 1,000 solicitudes concurrentes

3. **Pruebas de picos súbitos de carga.**

   Objetivo: Analizar la capacidad de adaptación del sistema a aumentos abruptos en la carga.

   Escenarios:

   - Incremento de 100 a 1,000 solicitudes en 20 segundos
   - Incremento de 200 a 2,000 solicitudes en 20 segundos

4. **Pruebas de estrés extremo.**

   Objetivo: Identificar el punto de saturación del sistema.

   Escenario:

   - 2,000 solicitudes concurrentes durante 5 minutos.
   - 3,000 solicitudes concurrentes hasta que el sistema falle.

Para visualizar todo el detalle completo del Plane de pruebas véase ¡Error! No se e ncuentra el origen de la referencia..

**Análisis completo del tiempo de respuesta y escalamiento**

**BookinCore**

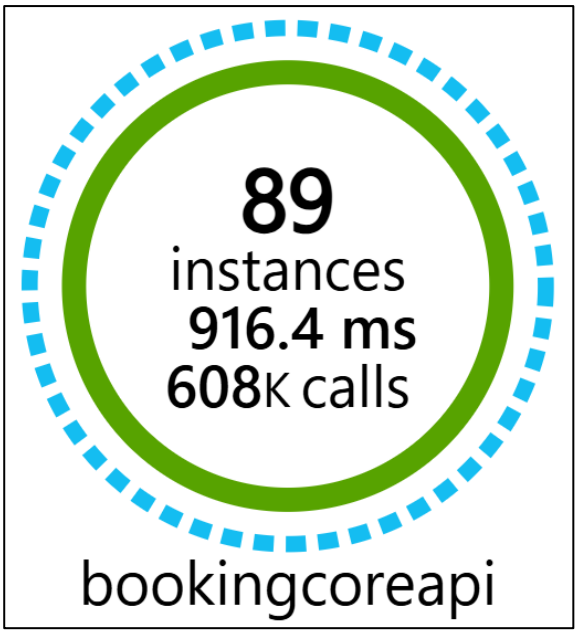



**Figura 29.** Registro de pruebas en BookingCore

Problemas de Escalabilidad:

- El límite de conexiones de la base de datos fue el principal factor que impidió que Azure Functions escalara efectivamente.
- El bajo uso de CPU (41%) muestra que los recursos disponibles no se utilizaron al máximo debido a restricciones de la base de datos.

Tiempos de Respuesta y Latencia:

- Los tiempos de respuesta excesivos, especialmente en ListEvents y ListSeatsByEvent, afectan la experiencia del usuario y reflejan un cuello de botella en el acceso a la base de datos.

Errores Críticos:

- La cantidad significativa de errores relacionados con conexiones no disponibles en la base de datos muestra la necesidad de mejorar su configuración y capacidad.

Las recomendaciones que se obtuvieron después de analizar BookingCore son:

1. **Optimización de la Base de Datos:**
    - Incrementar el límite de conexiones en PostgreSQL y habilitar el uso de conexiones reservadas (pg_use_reserved_connections).
    - Implementar un pool de conexiones para optimizar la gestión de recursos.
2. **Preescalamiento:**
    - Configurar instancias en Azure Functions para preescalar durante períodos de alta demanda prevista.
3. **Uso de Caché:**
    - Implementar un sistema de caché (como Redis) para reducir la dependencia de la base de datos en operaciones de lectura intensiva, como ListEvents y ListSeatsByEvent.
4. **Monitoreo Activo:**
    - Establecer alertas en Azure Monitor para identificar rápidamente problemas de escalabilidad y saturación de recursos.

**AuthCore**

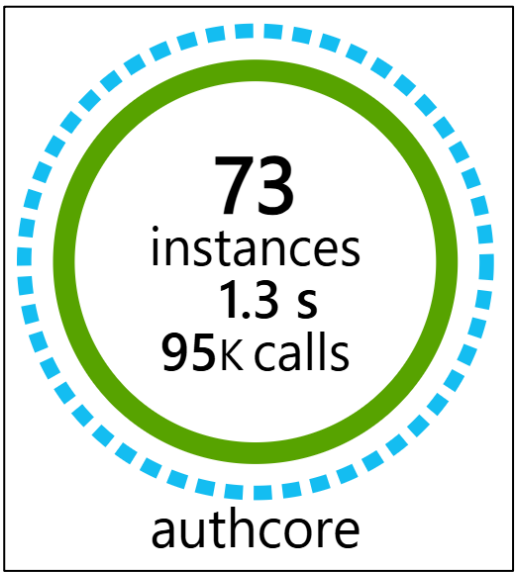



**Figura 30.** Registro de pruebas en AuthCore

Limitación en la Base de Datos:

- La causa principal de los errores y el bajo desempeño fue la falta de capacidad en la base de datos para manejar múltiples conexiones concurrentes.

Subutilización de Recursos:

- A pesar de un bajo uso de CPU, el sistema no pudo escalar adecuadamente debido a dependencias externas, como la base de datos.

Problemas de Red:

- Los errores de tiempo de espera indican posibles problemas de latencia o configuración en la comunicación entre el servicio y la base de datos.

Las recomendaciones que se obtuvieron después de analizar AuthCore son:

1. **Optimización de la Base de Datos:**
    - Incrementar el límite de conexiones y habilitar conexiones reservadas para roles privilegiados.
    - Implementar un pool de conexiones para manejar solicitudes concurrentes de manera más eficiente.
2. **Preescalamiento:**
    - Configurar instancias adicionales para manejar picos de carga de forma proactiva.
3. **Uso de Caché:**
    - Implementar caché para reducir la dependencia de la base de datos, especialmente para operaciones de lectura como ListUser.
4. **Monitoreo y Alertas:**
    - Configurar alertas en Azure Monitor para detectar rápidamente saturaciones en la base de datos y problemas de conectividad.
5. **Pruebas Adicionales:**
    - Realizar pruebas después de implementar estas optimizaciones para validar su efectividad y analizar métricas de mejora.

Para más detalles de los resultados de las pruebas de carga y estrés vistos desde el servicio de monitoreo de Azure llamado Application insights véase ¡Error! No se encuentra el o rigen de la referencia.

# 7. Discusión

## 7.1. Implementar la arquitectura Serverless con alta disponibilidad para la Gestión de reservas usando Azure mediante la metodología DevOps

La implementación de la arquitectura Serverless en Azure Functions marcó un hito clave en el desarrollo del sistema de gestión de reservas el cual fue el caso de estudio de este trabajo, asegurando alta disponibilidad y escalabilidad. Aunque desde el inicio se había planeado implementar una arquitectura serverless, durante las primeras etapas del desarrollo se optó por utilizar Express.js en un modelo tradicional basado en servidor. Esto se realizó bajo la suposición de que la migración hacia serverless en Azure Functions sería sencilla y no implicaría mayores complicaciones. Este enfoque inicial permitió avanzar rápidamente en el desarrollo y realizar pruebas funcionales básicas del sistema.

Sin embargo, al momento de implementar la solución en Azure Functions, se presentaron varios desafíos que hicieron evidente la necesidad de refactorizar el código previamente desarrollado. La arquitectura serverless de Azure Functions requiere un enfoque diferente al de un servidor tradicional, ya que las funciones operan en un modelo basado en eventos y deben ser diseñadas para ejecutarse de manera aislada y eficiente. Esto implicó reestructurar la lógica del sistema para adaptarla al entorno serverless, eliminando dependencias propias de un servidor persistente y optimizando el manejo de recursos y eventos.

Fue necesario modificar aspectos clave del código, como la configuración de las rutas, los controladores y las dependencias de base de datos, para que se alinearan con las características específicas de Azure Functions. A pesar de los desafíos, la decisión de refactorizar y migrar completamente hacia serverless fue altamente beneficiosa. Azure Functions permitió implementar un sistema que escala automáticamente según la demanda, reduciendo los costos operativos y garantizando una mayor flexibilidad.

La implementación de la arquitectura serverless basada en Azure Functions cumplió con el objetivo de garantizar servicios configurados y operativos, además de integrar el sistema de gestión de reservas con servicios de escalamiento automático. Este enfoque permitió observar los beneficios y desafíos asociados al desarrollo y despliegue de sistemas en la nube.

La configuración de Azure Functions como base serverless demostró ser efectiva para manejar las operaciones críticas del sistema, como la gestión de usuarios, roles, eventos y reservas.

- El sistema logró responder de manera efectiva a las solicitudes durante las pruebas funcionales.

- Se configuraron funciones HTTP con triggers que soportan operaciones clave del sistema, demostrando una operación estable y funcional en entornos simulados.

La integración de los módulos AuthCore y BookingCore con los servicios de escalamiento automático fue uno de los puntos más destacados del desarrollo. Este proceso implicó la conexión de los módulos con bases de datos en Azure Database for PostgreSQL - Flexible Server y la configuración de las funciones serverless para escalar dinámicamente con base en la carga. La arquitectura fue capaz de soportar múltiples solicitudes concurrentes.

La elaboración de una guía técnica detallada fue esencial para garantizar la reproducibilidad del sistema. Esta documentación incluyó:

- Pasos para configurar las funciones en Azure Functions y conectar los módulos con bases de datos.
- Diagramas que ilustran la arquitectura del sistema, destacando las interacciones entre los servicios y módulos.

Esta documentación no solo sirvió como referencia para los desarrolladores, sino que también establece un punto de partida para futuros mantenimientos o expansiones del sistema.

Entre los aspectos destacados tenemos que la metodología DevOps permitió una integración y despliegue fluido, maximizando la productividad, la configuración de los servicios y la integración del sistema demostraron ser robustas y alineadas con los requerimientos iniciales.

El primer objetivo fue cumplido con éxito, estableciendo una arquitectura serverless robusta y operativa en Azure Functions. La elección de Node.js resultó ser una decisión acertada, dadas sus ventajas para este tipo de entornos, como su eficiencia en operaciones concurrentes y su rapidez en tiempos de inicio. Los desafíos enfrentados durante la implementación proporcionaron aprendizajes valiosos, destacando la importancia de planificar detalladamente las configuraciones iniciales y de mantener una documentación técnica precisa para garantizar el éxito de proyectos futuros.

## 7.2. Determinar el tiempo de escalamiento usando pruebas de carga y estrés mediante pruebas de hipótesis

El análisis de los resultados obtenidos para los servicios AuthCore y BookingCore implementados en Azure Functions permitió evaluar la efectividad del escalamiento automático en un entorno serverless.

Aunque se desplegaron 81 instancias en AuthCore y 90 en BookingCore, el bajo uso de CPU (9% y 41%, respectivamente) evidencia que el sistema no alcanzó su potencial de escalabilidad debido a limitaciones externas, particularmente en la base de datos.

Los errores frecuentes, como "remaining connection slots are reserved" y "ECONNREFUSED", muestran que la infraestructura de la base de datos no estaba preparada para manejar la carga concurrente generada durante las pruebas.

Los errores relacionados con el código 499 (solicitudes canceladas por el cliente) son indicativos de tiempos de espera prolongados en las funciones, especialmente durante operaciones intensivas en lectura (ListEvents, ListSeatsByEvent).

Aunque el sistema mostró ciertas fortalezas, como tiempos de respuesta aceptables en operaciones menos intensivas (CreateUser, ListUser), los resultados resaltan que el desempeño general puede mejorarse significativamente al abordar los problemas relacionados con la base de datos y el escalamiento automático. Este análisis constituye una base para futuras iteraciones y optimizaciones en la arquitectura serverless.

Respondiendo a la pregunta de investigación de este trabajo de titulación, se concluye que el tiempo de escalamiento automático de un sistema de gestión de reservas con alta disponibilidad utilizando Azure Functions depende en gran medida de la configuración de la infraestructura y las condiciones de carga. Durante las pruebas realizadas, se observó que, aunque Azure Functions logró crear instancias adicionales para manejar la carga, el uso de CPU se mantuvo bajo (9% en AuthCore y 41% en BookingCore), lo que evidencia una subutilización de los recursos disponibles. Este resultado indica que, aunque el sistema puede escalar automáticamente, el tiempo efectivo de escalamiento está influenciado por cuellos de botella externos, como las limitaciones en la capacidad de la base de datos, lo que requiere ajustes en la configuración para garantizar un desempeño óptimo.

## 8. Conclusiones

- El sistema Serverless demostró un desempeño eficiente bajo cargas moderadas, registrando tiempos promedio de 548 ms en Login (AuthCore) y 752 ms en ListEvents (BookingCore) con tasas de error inferiores al 0.01%. Sin embargo, en escenarios de estrés extremo, se identificaron más de 10,721 errores de conexión y tiempos máximos de hasta 639,908 ms, lo que evidencia la necesidad de optimizar la configuración de la base de datos y considerar el uso de caché para garantizar un rendimiento adecuado en condiciones de alta demanda
- La generación de documentación técnica detallada sobre la implementación y configuración fue esencial para garantizar la replicabilidad del sistema y la comprensión de las configuraciones utilizadas. Durante el desarrollo, se subestimaron los desafíos técnicos asociados con la transición a un modelo serverless, lo que generó retrasos en la integración de servicios y ajustes de parámetros clave. Fue necesario realizar una refactorización completa del código para adaptarlo al modelo serverless, eliminando dependencias propias de un entorno de servidor persistente. Para solucionar estos problemas, se modularizó el código, se ajustó el manejo de solicitudes HTTP a un modelo basado en eventos. La documentación final consolidó estos cambios, proporcionando guías detalladas para futuras implementaciones y asegurando un despliegue eficiente bajo esta arquitectura.
- La implementación de la arquitectura serverless con Azure Functions logró establecer un sistema de alta disponibilidad y escalabilidad. La capacidad de escalamiento automático demostró ser eficaz en escenarios de carga moderada, garantizando la continuidad operativa sin tiempos de inactividad perceptibles. Sin embargo, se identificaron limitaciones en el manejo de recursos, especialmente en la base de datos, lo que destacó la necesidad de ajustes adicionales en la configuración del sistema para manejar escenarios de carga extrema de manera eficiente
- Una arquitectura serverless basada en Azure Functions y Node.js es una solución viable para sistemas críticos que requieren alta disponibilidad, escalabilidad y optimización de costos operativos. Sin embargo, también deja claro que la planificación y optimización continua son fundamentales para garantizar su éxito a largo plazo.

## 8.1. Trabajos Futuros

- Incorporar módulos adicionales al sistema, como reportes avanzados, análisis de datos en tiempo real o funcionalidades de personalización para los usuarios, podría aumentar el valor del sistema y ampliar su aplicabilidad en diferentes contextos industriales.
- Aunque Azure Functions fue la tecnología elegida, futuros trabajos podrían comparar su desempeño con otras soluciones serverless, como AWS Lambda o Google Cloud Functions.
- Investigar y probar configuraciones más agresivas de escalamiento automático en Azure Functions, como el ajuste de umbrales de CPU y memoria, para reducir los tiempos de escalamiento en servicios con cargas más complejas, como **BookingCore**.
- Evaluar la posibilidad de migrar a servicios adicionales de Azure que complementen la solución actual, como Azure Cosmos DB para manejar datos no relacionales o sistemas híbridos de bases de datos que permitan mayor flexibilidad y rendimiento.
- Evaluar el uso de un sistema de caché distribuido (como Redis) para reducir la carga en operaciones intensivas de lectura, como ListEvents y ListSeatsByEvent.
- Implementar un sistema de monitoreo más robusto que permita detectar y alertar de manera anticipada problemas relacionados con el escalamiento, uso de recursos y latencia en la base de datos.

## 9. Recomendaciones

- Es recomendable que las herramientas utilizadas, como Azure Functions y Node.js, se mantengan actualizadas para garantizar compatibilidad, seguridad y acceso a las últimas funcionalidades. Esto es crucial para optimizar el desempeño del sistema y prevenir posibles vulnerabilidades.
- Se recomienda configurar herramientas como Azure Monitor y Application Insights para monitorear continuamente el rendimiento del sistema. Esto permitirá detectar problemas de forma proactiva, evaluar la efectividad del escalamiento automático y realizar ajustes según sea necesario.
- Si bien Azure Functions demostró ser una solución efectiva, se recomienda investigar y probar configuraciones avanzadas de escalamiento automático, como el uso de triggers personalizados basados en métricas de uso específicas, para lograr una mayor optimización de recursos.
- Mantener una documentación actualizada de las pruebas realizadas, resultados obtenidos y configuraciones utilizadas, asegurando la transferencia de conocimiento para futuras iteraciones del proyecto.
- Se recomienda configurar preescalamiento en Azure Functions para garantizar que las instancias estén listas antes de los picos de carga previstos. Esto reducirá los tiempos de inicio y mejorará la capacidad de respuesta del sistema en condiciones de alta concurrencia.
- Utilizar un pool de conexiones para optimizar el manejo de solicitudes a la base de datos, asegurando un uso más eficiente de los recursos disponibles y evitando saturaciones en operaciones concurrentes.
- Realizar un análisis periódico de los costos asociados a la arquitectura serverless, incluyendo el consumo de Azure Functions, bases de datos y almacenamiento.
- Desarrollar un conjunto de pruebas automatizadas que cubran los principales flujos del sistema, asegurando que cada cambio o actualización pase por validaciones estrictas antes de ser desplegado

## 10. Bibliografía


[1] J. Smith and E. Brown, “Cloud scalability and its application in resource-intensive systems,” *Journal of Cloud Computing*, vol. 9, pp. 123–135, 2020.

[2] M. Jones and S. Miller, “Using Kubernetes for dynamic load management in web applications,” *Software Engineering Trends*, vol. 15, pp. 45–58, 2021.

[3] K. Venkatesan and E. Siddharth, “Optimizing Serverless Architectures for High-Throughput Systems Using AWS Lambda and DynamoDB,” *IEEE Transactions on Cloud Computing*, vol. 8, no. 2, pp. 413–426, Feb. 2025.

[4] E. A. Arevalo Chavez, “Infografia de recursos tecnologicos.” Accessed: Feb. 22, 2025. [Online]. Available: https://infogram.com/infografia-de-recursos-tecnologicos-1hnp27m3qlnvy2g

[5] E. Brewer, Y. Sheng, and K. Veeraraghavan, “Evaluation of Serverless Cloud Computing Services,” *Proceedings of the 22nd International Conference on Cloud Computing and Big Data (CLOUD)*, 2021.

[6] I. Baldini *et al.*, “Serverless computing: Current trends and open problems,” in *Research Advances in Cloud Computing*, 2017, pp. 1–20.

[7] H. Shafiei, A. Khonsari, and P. Mousavi, “Serverless Computing: A Survey of Opportunities, Challenges, and Applications,” *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 9, no. 1, pp. 1–23, 2020.

[8] R. Ranjan, “Serverless architecture for large-scale online reservation systems,” *IEEE Trans Serv Comput*, vol. 12, no. 5, pp. 780–792, 2019.

[9] M. Armbrust *et al.*, “A view of cloud computing,” *Commun ACM*, vol. 53, no. 4, pp. 50–58, 2010.

[10] Q. Zhang, L. Cheng, and R. Boutaba, “Cloud computing: state-of-the-art and research challenges,” *Journal of internet services and applications*, vol. 1, no. 1, pp. 7–18, 2010.

[11] I. A. Targio Hashem, Y. Ibrar, N. B. Anuar, S. Mokhtar, A. U. Gani, and S. Khan, “The rise of ‘big data’ on cloud computing: Review and open research issues,” *Inf Syst*, vol. 47, pp. 98–115, 2015.

[12] P. Mell and T. Grance, *The NIST Definition of Cloud Computing*. NIST, 2011.

[13] T. Yukio, “Cloud native application architectures: Current state and future directions,” *IEEE Transactions on Cloud Computing*, vol. 9, no. 3, pp. 1315–1331, 2020.

[14] E. S. Shekhar, “Evaluating Scalable Solutions: A Comparative Study of AWS, Azure, and GCP,” vol. 6, pp. 2456–4184, 2021, Accessed: Feb. 23, 2025. [Online]. Available: www.ijnrd.org

[15] G. Adzic and R. Chatley, “Serverless computing: economic and architectural impact,” *IEEE Cloud Computing*, vol. 6, no. 6, pp. 32–39, 2019.

[16] Microsoft, “Azure Kubernetes Service (AKS) | Microsoft Azure.” Accessed: Feb. 23, 2025. [Online]. Available: https://azure.microsoft.com/es-es/products/kubernetes-service

[17] A. Agache *et al.*, “Firecracker: Lightweight virtualization for serverless applications,” in *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, 2019, pp. 419–434.

[18] V. K. Vavilapallih *et al.*, *Apache Hadoop YARN: Yet Another Resource Negotiator*. ACM, 2013.

[19] P. Sharma, S. Lee, and M. Othman, “Serverless Computing: A Review and Research Agenda,” *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 9, no. 1, pp. 1–23, 2020.

[20] Microsoft, “Equilibrio de carga para varias regiones - Azure Reference Architectures | Microsoft Learn.” Accessed: Feb. 23, 2025. [Online]. Available: https://learn.microsoft.com/es-es/azure/architecture/high-availability/reference-architecture-traffic-manager-application-gateway

[21] M. Netto, C. Cardonha, R. Cunha, and M. Assuncao, “Evaluating Auto-scaling Strategies for Cloud Computing Environments,” *Journal of Cloud Computing: Advances, Systems and Applications*, vol. 10, no. 1, pp. 1–16, 2021.

[22] J. Smith and E. Brown, “Performance Analysis of Serverless Architectures with Node.js, Python, Go, and Java,” *Journal of Cloud Computing*, vol. 12, pp. 134–150, 2021.

[23] M. Jones and S. Miller, “Concurrency Models in Serverless Computing: A Comparative Study,” *Proceedings of the ACM on Serverless Computing*, vol. 8, pp. 22–30, 2020.

[24] R. M. Llontop Garcia, “Implementación de una arquitectura escalable basada en Google Cloud Platform para mejorar la disponibilidad y escalabilidad de información de la

empresa SmartBrands, Lima 2019," 2020. [Online]. Available: https://tesis.usat.edu.pe/handle/20.500.12423/2921

[25] M. D. Llauce Santos, "Implementación de una arquitectura de computación en la nube (Cloud Computing) diseñada para escalabilidad automática y alta disponibilidad basado en la plataforma de Amazon Web Services (AWS) en la Universidad de Lambayeque," 2020. [Online]. Available: https://repositorio.unprg.edu.pe/handle/20.500.12893/10132

[26] E. Sanfrutos, "Aplicación de tecnologías y arquitecturas serverless para el desarrollo de soluciones IoT," 2020. [Online]. Available: https://openaccess.uoc.edu/bitstream/10609/116206/8/esanfrutosTFM0620memoria.pdf

[27] J. N. Mayanquer Rosero, "Análisis comparativo de las arquitecturas serverless, que los diferentes proveedores, ofrecen en la nube orientado a la creación de una aplicación web," 2022. [Online]. Available: https://repositorio.puce.edu.ec/items/0ca1df13-d539-437f-b22d-5c4367291a3c

# 11. Anexos

**Anexo 1.** Repositorios de Código de Servicios para la Gestión de Reservas

## AuthCore

https://github.com/omarAlexis1999/authCore.git

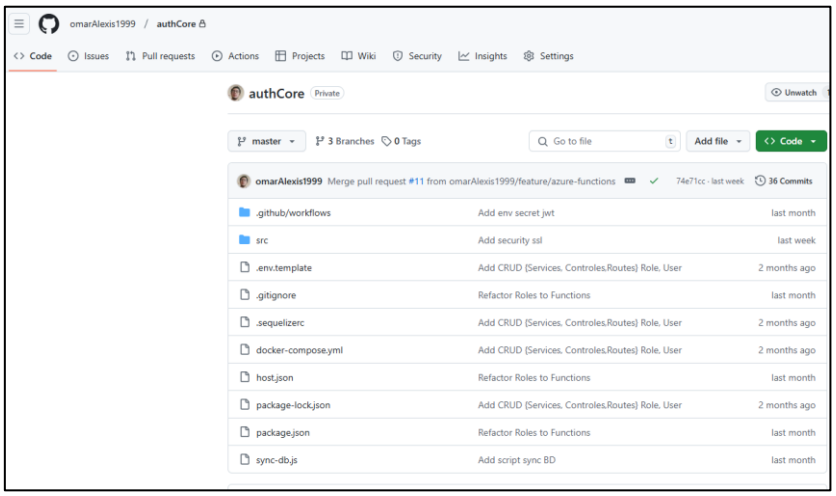



**Figura 31.** Repositorio de Github AuthCore

## BookinCore

https://github.com/omarAlexis1999/bookingCore.git

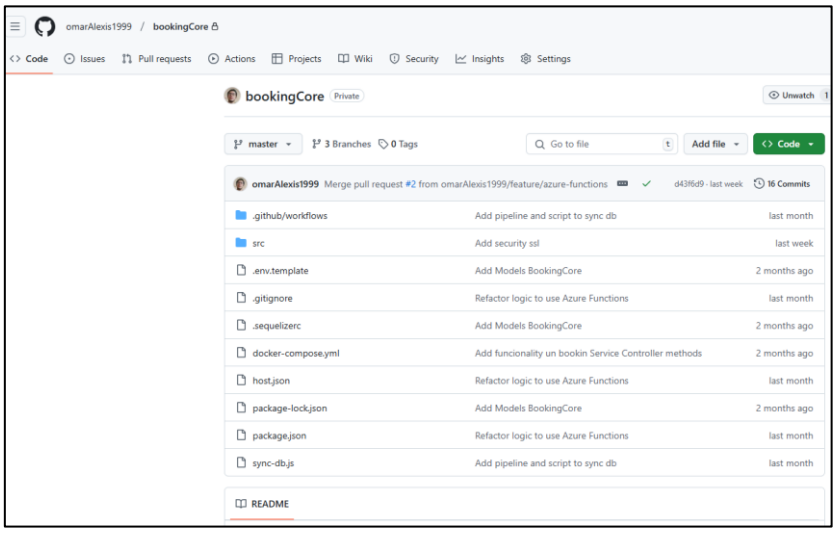



**Figura 32.** Repositorio de Github BookingCore

**Anexo 2.** Dirección de URL de servicios AuthCore y BookingCore

### Servicios Functions App

- **AuthCore**: authcore.azurewebsites.net
- **BookingCore**: bookingcoreapi.azurewebsites.net

### Gateway:

- https://gatewaygestionreservas.azure-api.net/
    - **AuthCore**: https://gatewaygestionreservas.azure-api.net/authCore
    - **BookingCore**: https://gatewaygestionreservas.azure-api.net/bookingCoreApi

## Manual del Proceso de Implementación y Configuración de Función App

Este manual detalla el proceso de implementación y configuración de una arquitectura **serverless** utilizando **Azure Functions** para soportar un sistema de gestión de reservas. Está diseñado para proporcionar una guía clara y reproducible que permita a los desarrolladores configurar, desplegar y operar una solución escalable y de alta disponibilidad.

El documento cubre desde la preparación del entorno local de desarrollo hasta el despliegue en la nube y las pruebas finales. Además, incluye prácticas recomendadas para garantizar la seguridad de las configuraciones, como el uso de estrategias de monitoreo con **Application Insights** para evaluar el desempeño del sistema.

Este manual está orientado a proyectos que requieran escalabilidad dinámica y tiempos de respuesta bajos, destacando las ventajas de utilizar Node.js en entornos serverless. Al final del proceso, el lector contará con un sistema funcional, seguro y preparado para manejar escenarios de alta concurrencia.

### Arquitectura

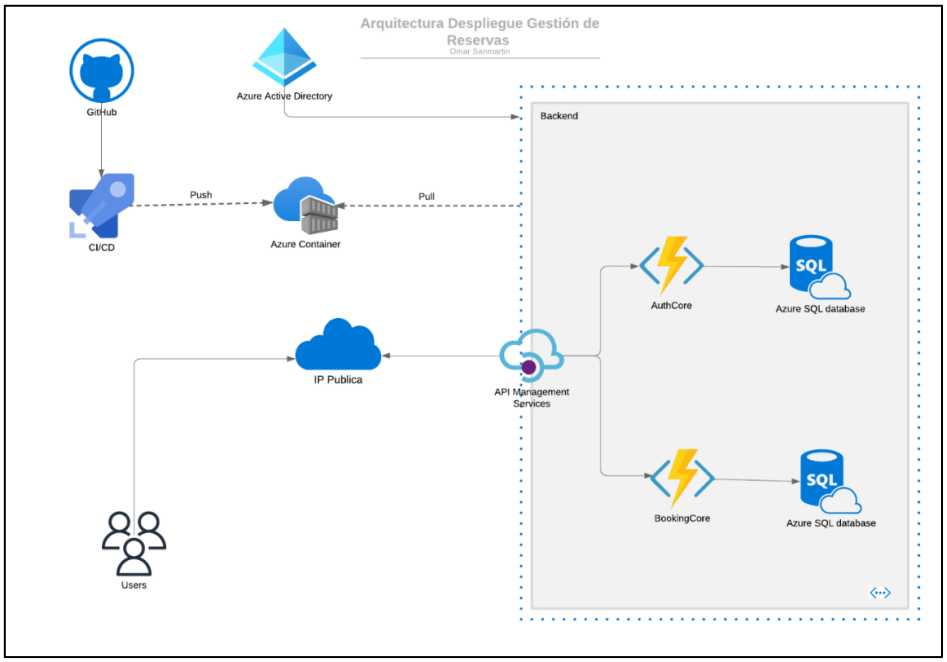



**Figura 33.** Arquitectura de Aplicción Gestión de Reservas

Contenido

## 1. Configuración del Entorno Local

Antes de desplegar la aplicación en **Azure Functions**, se configuró un entorno local para desarrollar y probar la aplicación. Los pasos principales son los siguientes:

**Instalación de Dependencias:**

1. **Node.js**: Asegúrate de tener instalada la versión compatible (recomendado ≥14.x).
2. **Azure Functions Core Tools**: Instálalo globalmente para trabajar localmente con funciones serverless.
    a. *npm install -g azure-functions-core-tools@4 --unsafe-perm true*

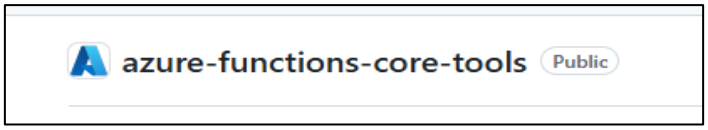



**Figura 34.** Libreria Azure Functions Core Tools

3. Instale Docker para un entorno de base de datos local.
    a. Puede instalarla haciendo uso de Docker Compose, esta configuración también incluirá un Gestor de Base de Datos

```yaml
version: '3.8'
services:
  postgres:
    image: postgres:latest
    container_name: authcore-db
    environment:
      POSTGRES_USER: sanmartinomar
      POSTGRES_PASSWORD: sanmartinomar
      POSTGRES_DB: authcore_db
    ports:
      - "5432:5432"
    networks:
      - authcore-network
    volumes:
      - C:/Postgres/authcore:/var/lib/postgresql/data

  pgadmin:
    image: dpage/pgadmin4
    container_name: pgadmin4
    environment:
      PGADMIN_DEFAULT_EMAIL: admin@admin.com
      PGADMIN_DEFAULT_PASSWORD: admin
    ports:
      - "8080:80"
    depends_on:
      - postgres
    networks:
      - authcore-network

networks:
  authcore-network:
    driver: bridge
```

**Figura 35.** Archivo Docker Compose

b. Una vez instalada podrá tener una base de datos de Postgres de manera local para pruebas.


**Figura 36.** Base de datos creada en Docker

**Inicialización del Proyecto:**

1. Cree un directorio para el proyecto y accede a él:
    a. mkdir myFunctionApp && cd myFunctionApp
2. Inicialice el proyecto:
    a. *func init reservationSystem --worker-runtime node*
3. Cree las funciones necesarias:
    a. func new
4. Como, por ejemplo, para gestionar usuarios y reservas:
    a. *func new --template "HTTP trigger" --name AuthCore*
    b. *func new --template "HTTP trigger" --name BookingCore*

**Estructura del Proyecto:**

1. Organiza los archivos y módulos Node.js de la aplicación para que las funciones estén encapsuladas y sigan las mejores prácticas. Una estructura típica incluye:

    - myFunctionApp/
    - ├── host.json
    - ├── local.settings.json
    - ├── package.json
    - ├── .gitignore
    - ├── src/
    - │ ├── functions/
    - │ ├── services/
    - │ ├── models/
    - │ ├── routes/
    - │ ├── utils/
    - │ └── middlewares/

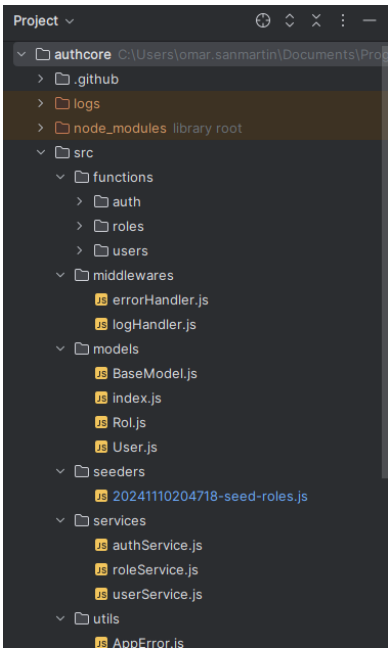



**Figura 37.** Estructura de Proyecto Function App de Azure

**Pruebas Locales:**

1. Configure las variables en local.settings.json:

```
{
  "IsEncrypted": false,
  "Values": {
    "FUNCTIONS_WORKER_RUNTIME": "node",
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "DB_HOST": "localhost",
    "DB_USER": "user",
    "DB_PASSWORD": "password",
    "DB_NAME": "database"
  }
}
```

2. Usa el comando func start para ejecutar y probar las funciones localmente:

    a. *func start*

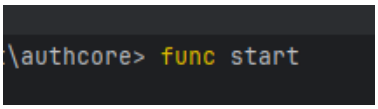



**Figura 38.** Ejecución de Function App de manera local

3. Verifica los endpoints generados, típicamente disponibles en:

    a. *http://localhost:7071*


**Figura 39.** Ejecución de Function App

## 2. Configuración en Azure

### Creación del Recurso Azure Function App:

1. Accede al **portal de Azure** y crea un nuevo recurso de tipo **Function App**.

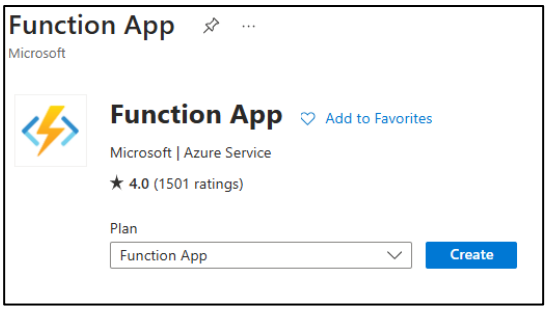



**Figura 40.** Instalación de recurso Function App en Azure

2. Cree un recurso Function App, configura los siguientes parámetros:

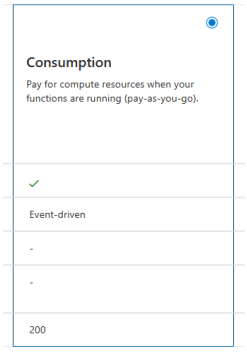



**Figura 41.** Plan de Consumption de Azure Function

- Sistema operativo: Linux
- Plan de hospedaje: Consumption (Serverless)
- Tiempo de ejecución: Node.js 18 LTS

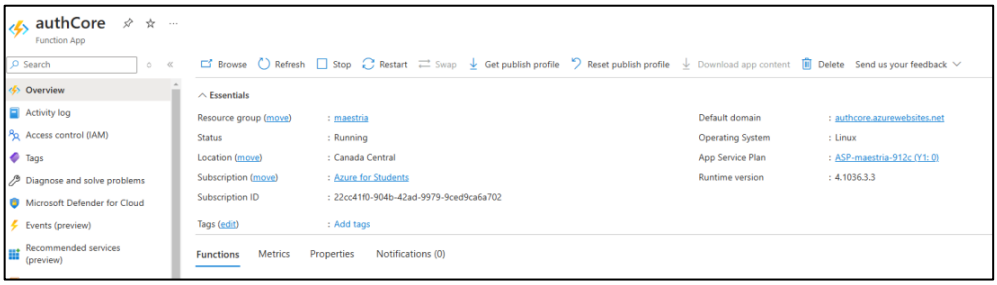



**Figura 42.** Ejecución de aplicación AuthCore (Sin servidor)

**Creación de API Management service (Gateway):**

1. Accede al **portal de Azure** y crea un nuevo recurso de tipo **API Management service**.

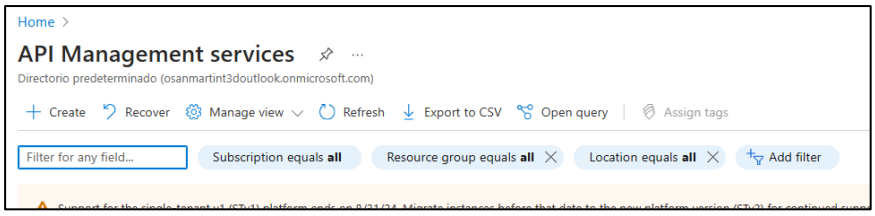



**Figura 43.** API Management services

2. Desde **API Management service** agregue el recurso Function AuthCore y BookingCore:

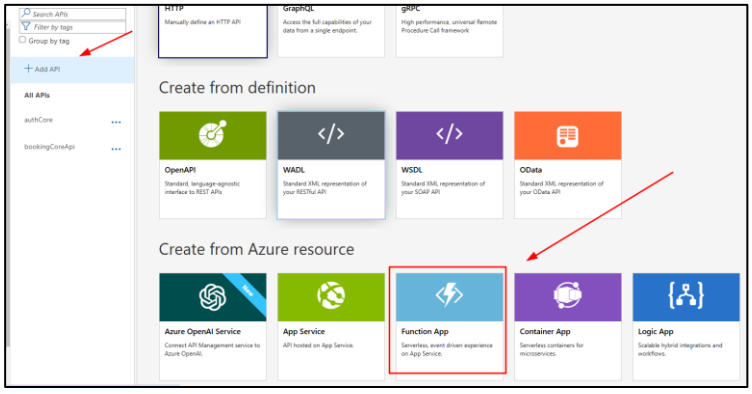



**Figura 44.** Agregar Function App en Gateway

3. Configure todas las políticas necesarias de para el acceso a los recursos de la aplicación, puede hacerlo por cada endpoint.

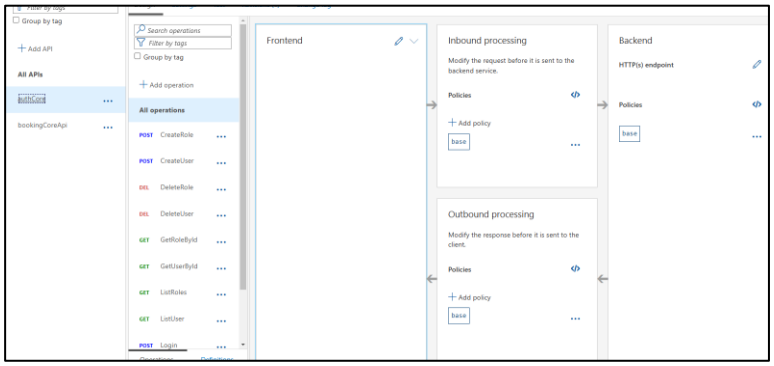



**Figura 45.** Configuración de politicas API Management services

**Configuración de la Base de Datos:**

1. Crea un recurso **PostgreSQL flexible en Azure** para almacenar los datos relacionados con usuarios, roles, eventos y reservas.

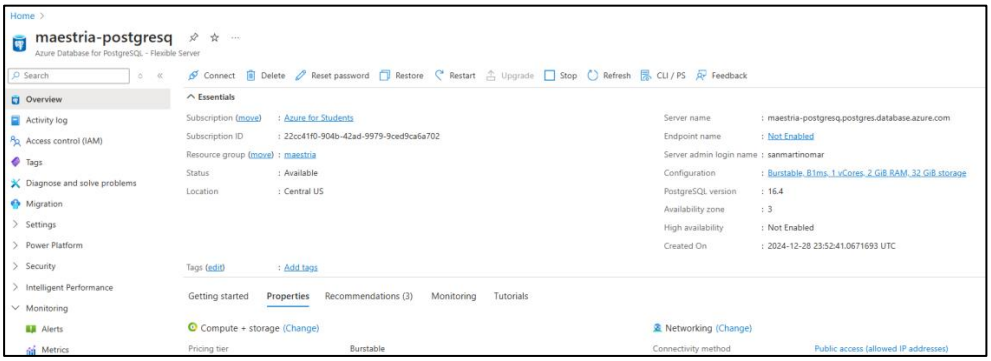



**Figura 46.** Creación de base de datos Postgres

2. Configure las reglas de red para permitir conexiones locales.

### 3. Despliegue de la Aplicación

**Preparación del Proyecto:**

1. Asegúrate de tener un archivo local.settings.json que defina las variables de entorno utilizadas localmente:

```
{
  "IsEncrypted": false,
  "Values": {
    "AzureWebJobsStorage": "UseDevelopmentStorage=true",
    "FUNCTIONS_WORKER_RUNTIME": "node",
    "DATABASE_URL": "<cadena de conexión de la base de datos>"
  }
}
```

**Publicación desde CLI:**

1. Inicie sesión en Azure:
    - az login
2. Usa el comando func azure functionapp publish para desplegar la aplicación:
    - *func azure functionapp publish <NombreDeTuFunctionApp>*

**Automatización con GitHub Actions (Opcional):**

1. Crea un archivo *.github/workflows/deploy.yml* para automatizar el despliegue con CI/CD.

```yaml
jobs:
  deploy:
    environment: DEV

    steps:
      - name: Checkout Code
        uses: actions/checkout@v3

      - name: Setup Node ${{ vars.NODE_VERSION }} Environment
        uses: actions/setup-node@v3
        with:
          node-version: ${{ vars.NODE_VERSION }}

      - name: 'Azure Login'
        uses: Azure/login@v2
        with:
          creds: ${{ secrets.AZURE_CREDENTIALS }}

      - name: 'Resolve Project Dependencies Using Npm'
        shell: bash
        run: |
          pushd './${{ vars.AZURE_FUNCTIONAPP_PACKAGE_PATH }}'
          npm install
          npm run build --if-present
          popd

      - name: 'Sync Database'
        env:
          DB_NAME: ${{ vars.DB_NAME }}
          DB_USER: ${{ secrets.DB_USER }}
          DB_PASSWORD: ${{ secrets.DB_PASSWORD }}
          DB_HOST: ${{ vars.DB_HOST }}
          DB_PORT: ${{ vars.DB_PORT }}
        run: node sync-db.js

      - name: 'Run Azure Functions Action'
        uses: Azure/functions-action@v1
```

**Figura 47.** Pipeline para despliegue de Function App en Azure

## 4. Validación y Monitoreo

### Pruebas Funcionales en Azure:

1. Verifica los endpoints en la URL proporcionada por Azure Function App, como:
    - https://<NombreDeTuFunctionApp>.azurewebsites.net/api/AuthCore

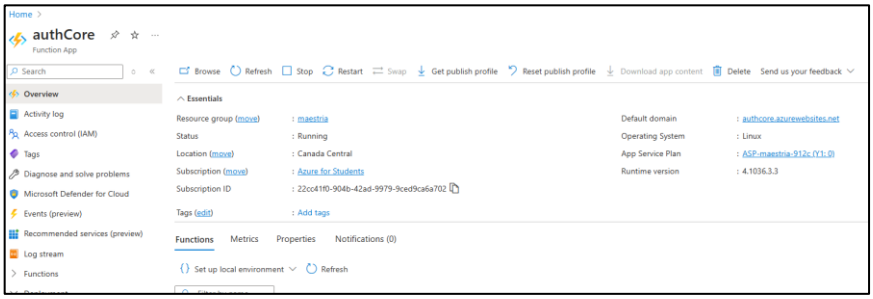



**Figura 48.** AuthCore aplicación sin servidor

2. Use Postman o curl para probar los endpoints desplegados.

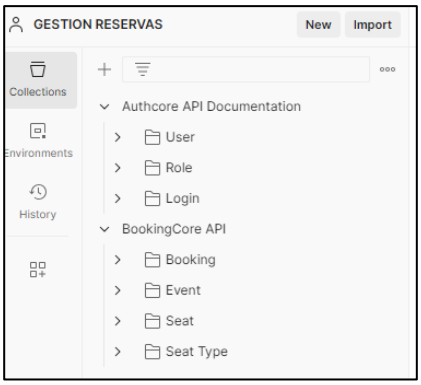



**Figura 49.** Colección de Postman con servicios de Gestión de Reservas

**Monitoreo del Desempeño:**

1. Acceda a Application Insights desde el recurso Function App.
2. Monitoree:
    - Llamadas exitosas y fallidas
    - Tiempo de ejecución promedio
    - Errores registrados con context.log.

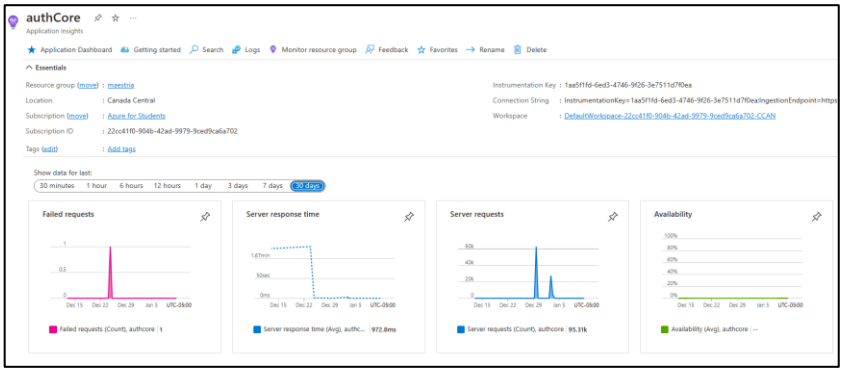



**Figura 50.** Application Insights de AuthCore (Function App)

## 5. Conclusión

Se ha configurado e implementado correctamente una arquitectura serverless utilizando Azure Function App. La aplicación está diseñada para ser escalable, eficiente y modular, cumpliendo con los requisitos del sistema de gestión de reservas. La integración con herramientas de CI/CD y monitoreo asegura su mantenibilidad y rendimiento a largo plazo.

**Anexo 4.** Plan de Pruebas Gestión de Reservas

# Plan de Pruebas Gestión de Reservas

## Introducción

Este documento describe el plan de pruebas para evaluar el desempeño de los servicios **AuthCore** y **BookingCore** implementados en **Azure Functions**. Las pruebas se enfocan en escenarios de carga y estrés para garantizar la escalabilidad, estabilidad y capacidad de manejo de solicitudes concurrentes.

## Objetivo de las Pruebas

El objetivo es analizar el comportamiento del sistema bajo diferentes niveles de carga y estrés, midiendo:

4. **Tiempo de respuesta:** La rapidez con la que los servicios procesan solicitudes.
5. **Tiempo de escalamiento:** El tiempo necesario para que Azure Functions cree nuevas instancias durante picos de carga.
6. **Estabilidad:** La capacidad del sistema para mantener la operatividad sin errores durante escenarios de alta demanda.

## Alcance

### Servicios Evaluados:

- **AuthCore:** Encargado de la autenticación y gestión de usuarios.
- **BookingCore:** Encargado de la gestión de eventos, asientos y reservas.

### Pruebas Incluidas:

5. Pruebas de carga progresiva.
6. Pruebas de carga sostenida.
7. Pruebas de picos súbitos de carga.
8. Pruebas de estrés extremo.

**Herramientas y Recursos**

- **Herramientas:**
    - Apache JMeter para la generación de cargas y simulación de usuarios concurrentes.
    - Azure Monitor y Application Insights para la recolección de métricas como tiempo de respuesta, tasa de éxito y número de instancias activas.
- **Recursos:**
    - Infraestructura: Azure Functions (Plan de Consumo).
    - Base de datos: Azure Database for PostgreSQL - Flexible

**Estrategia de Pruebas**

**Tipos de Pruebas**

1. **Pruebas de Carga:**
    - Incrementar progresivamente el número de solicitudes concurrentes.
    - Medir tiempos de respuesta y tasa de éxito.
2. **Pruebas de Estrés:**
    - Simular escenarios extremos con un número de solicitudes superior a la capacidad teórica.
    - Identificar el punto de fallo del sistema y el tiempo de recuperación.
3. **Pruebas de Escalamiento Automático:**
    - Medir el tiempo que tarda Azure Functions en crear nuevas instancias.
    - Verificar la estabilidad del sistema tras el escalamiento.

**Casos de Prueba**

**1. Prueba de Carga Progresiva**

- **Objetivo:** Evaluar el tiempo de respuesta y la tasa de éxito al incrementar progresivamente las solicitudes.
- **Métrica Clave:** Tiempo promedio de respuesta por nivel de carga.
- **Escenarios:**
    - 200 solicitudes concurrentes.
        - Numero de Hilos: 200
        - Periodo de Subida: 50
        - Usuarios por segundo: 4
    - 500 solicitudes concurrentes.
        - Numero de Hilos: 500
        - Periodo de Subida: 50
        - Usuarios por segundo: 10
    - 1,000 solicitudes concurrentes.
        - Numero de Hilos: 1000
        - Periodo de Subida: 50
        - Usuarios por segundo: 20

**2. Prueba de Carga Sostenida**

- **Objetivo:** Verificar la estabilidad del sistema durante periodos prolongados de carga constante.
- **Duración:** 15 minutos por nivel de carga.
- **Niveles de Carga:**
    - 500 solicitudes concurrentes.
        - Numero de Hilos: 500
        - Periodo de Subida: 100
        - Usuarios por segundo: 5
        - Duración 900 segundos (15 minutos)
        - Loop Count: Infinito
    - 1,000 solicitudes concurrentes.
        - Numero de Hilos: 1000
        - Periodo de Subida: 100
        - Usuarios por segundo: 10
        - Duración 900 segundos (15 minutos)
        - Loop Count: Infinito

**3. Prueba de Picos Súbitos**

- **Objetivo:** Analizar la capacidad de adaptación del sistema a aumentos abruptos en la carga.
- **Escenarios:**
    - Incremento de 100 a 1,000 solicitudes en 20 segundos.
        - Primera Grupo de Usuarios
            - Numero de Hilos: 100
            - Periodo de Subida: 20
            - Usuarios por segundo: 5
            - Duración 20 segundos
        - Segundo Grupo de Usuarios
            - Numero de Hilos: 1000
            - Periodo de Subida: 20
            - Usuarios por segundo: 50

        - Duración 20 segundos
    - Incremento de 200 a 2000 solicitudes en 20 segundos.
        - Primera Grupo de Usuarios
            - Numero de Hilos: 200
            - Periodo de Subida: 20
            - Usuarios por segundo: 10
            - Duración 20 segundos
        - Segundo Grupo de Usuarios
            - Numero de Hilos: 2000
            - Periodo de Subida: 20
            - Usuarios por segundo: 100
            - Duración 20 segundos

**4. Prueba de Estrés Extremo**

- **Objetivo:** Identificar el punto de saturación del sistema.
- **Escenario:**
    - 2,000 solicitudes concurrentes durante 5 minutos.
        - Numero de Hilos: 2000
        - Periodo de Subida: 20
        - Usuarios por segundo: 100
        - Duración 300 segundos (5 minutos)
        - Loop Count: Infinito
    - 3,000 solicitudes concurrentes hasta que el sistema falle.
        - Numero de Hilos: 3000
        - Periodo de Subida: 15
        - Usuarios por segundo: 50
        - Duración 300 segundos (5 minutos)
        - Loop Count: Infinito

**Métricas a Evaluar**

1. **Tiempo de Respuesta:** Promedio, mínimo y máximo por nivel de carga.
2. **Tasa de Éxito:** Porcentaje de solicitudes completadas sin errores.
3. **Tiempo de Escalamiento:** Tiempo desde el aumento de la carga hasta la estabilización del sistema.
4. **Número de Instancias Activas:** Cantidad de instancias creadas durante picos de carga.
5. **Uso de Recursos:** Uso promedio de CPU y memoria por instancia.

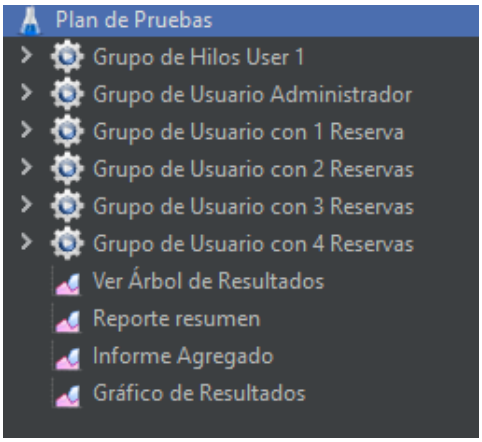



**Figura 51.** Hilo de usuarios Jmeter

**Cronograma de Ejecución**

1. **Semana 1:** Configuración de herramientas y definición de escenarios.
2. **Semana 2:** Ejecución de pruebas de carga progresiva y sostenida.
3. **Semana 3:** Ejecución de pruebas de picos súbitos y estrés extremo.
4. **Semana 4:** Análisis de datos y generación del informe.

## Informe de Resultados

## Prueba de Carga Progresiva

- 200 solicitudes concurrentes.

| Etiqueta | # Muestras | Media | Mín | Máx | Desv. Estándar | % Error | Rendimiento | Kb/sec | Sent KB/sec | Media de Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 50 | 603 | 0 | 3003 | 455,54 | 0 | 1,00835 | 7,33 | 0,18 | 7444,6 |
| Listar Asientos | 50 | 240 | 0 | 2116 | 272,13 | 0 | 1,07381 | 2,22 | 0,25 | 2118,7 |
| Login | 50 | 1022 | 0 | 4784 | 1140,22 | 0 | 1,07246 | 0,49 | 0,31 | 464,5 |
| Registrar Reserva | 50 | 295 | 0 | 507 | 62,18 | 0 | 1,13525 | 0,43 | 0,54 | 385,8 |
| Total | 200 | 540 | 0 | 4784 | 702,18 | 0 | 3,95398 | 10,05 | 1,15 | 2603,4 |

**Figura 52.** Resultados Pruebas de 200 solicitudes

  - El tiempo de respuesta promedio más alto fue de 1,022 ms (Login).
  - La operación de "Registrar Reserva" tuvo el mejor desempeño promedio con 295 ms.
  - Ningún tiempo máximo excedió los 5 segundos, lo que indica que el sistema manejó esta carga con estabilidad.

- 500 solicitudes concurrentes.

| Etiqueta | # Muestras | Media | Mín | Máx | Desv. Estándar | % Error | Rendimiento | Kb/sec | Sent KB/sec | Media de Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 500 | 752 | 0 | 4960 | 526,76 | 0 | 9,96373 | 72,44 | 1,73 | 7445,2 |
| Listar Asientos | 500 | 299 | 0 | 2077 | 217,77 | 0,002 | 10,61458 | 21,67 | 2,48 | 2090,1 |
| Login | 500 | 3958 | 0 | 16348 | 5110,81 | 0 | 10,67486 | 4,84 | 3,08 | 463,9 |
| Registrar Reserva | 375 | 498 | 0 | 2197 | 379,46 | 0 | 8,64075 | 3,24 | 3,96 | 384,2 |
| Total | 1875 | 1435 | 0 | 16348 | 3069,65 | 0,00053 | 36,69204 | 98,3 | 10,17 | 2743,3 |

**Figura 53.** Resultados Pruebas de 500 solicitudes

  - El tiempo de respuesta promedio para "Login" aumentó significativamente a 3,958 ms, con un máximo de 16,348 ms, lo cual es crítico para la experiencia del usuario.
  - La operación "Listar Asientos" mantuvo un tiempo promedio bajo de 299 ms, lo que demuestra buena optimización.

- 1,000 solicitudes concurrentes.

| Etiqueta | # Muestras | Media | Mín | Máx | Desv. Estándar | % Error | Rendimiento | Kb/sec | Sent KB/sec | Media de Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 1000 | 882 | 0 | 21301 | 874,66 | 0,085 | 18,4478 | 123,18 | 3,2 | 6837,7 |
| Listar Asientos | 1000 | 470 | 0 | 2295 | 373,06 | 0,084 | 19,41333 | 36,97 | 4,53 | 1949,8 |
| Login | 1000 | 10752 | 0 | 27843 | 9270,08 | 0,047 | 19,59094 | 8,72 | 5,65 | 455,9 |
| Registrar Reserva | 750 | 1055 | 0 | 5314 | 780,71 | 0,21733 | 15,84117 | 5,43 | 7,15 | 350,9 |
| Total | 3750 | 3439 | 0 | 27843 | 6540,09 | 0,10107 | 67,70786 | 167,62 | 18,67 | 2535,1 |

**Figura 54.** Resultados Pruebas de 1000 solicitudes

  - El tiempo promedio de "Login" fue el más alto con 10,752 ms, y el tiempo máximo llegó a 27,843 ms. Esto indica que esta operación se convierte en un cuello de botella bajo alta carga.
  - "Registrar Reserva" mantuvo un tiempo promedio aceptable de 1,055 ms, pero con un tiempo máximo de 5,314 ms, lo que sugiere la necesidad de optimización para escenarios de carga extrema.

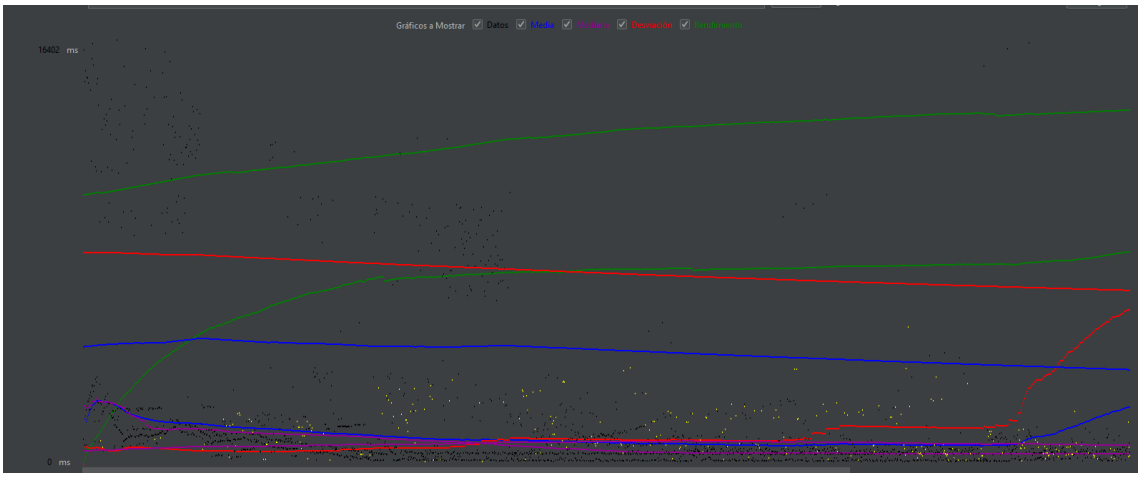



**Figura 55.** Resumen de pruebas de carga de 1000 solicitudes

### Observaciones Generales

Login es la operación más sensible al incremento de carga, con un aumento desproporcionado en los tiempos de respuesta.

Listar Eventos y Listar Asientos muestran un comportamiento estable hasta 500 solicitudes concurrentes, pero su desempeño se degrada con 1,000 solicitudes.

Se observaron tasas de error más altas en "Registrar Reserva" (21.7%) y "Listar Asientos" (8.4%). Esto sugiere que el sistema comienza a saturarse y no puede manejar adecuadamente las solicitudes.

- 200 solicitudes: Rendimiento total de 3.95 solicitudes/segundo y 10.05 KB/seg enviados.
- 500 solicitudes: Rendimiento total de 36.69 solicitudes/segundo y 98.3 KB/seg enviados.
- 1,000 solicitudes: Rendimiento total de 67.7 solicitudes/segundo y 167.62 KB/seg enviados.

A pesar del aumento en la carga, el rendimiento sigue siendo lineal, pero las operaciones como "Login" tienen problemas de escalabilidad.

**Prueba de Carga Sostenida**

- 500 solicitudes concurrentes.

| Etiqueta | # Muestras | Media | Mín | Máx | Desv. Est | % Error | Ren | Kb/s | Sent KB | Media de Byte |
|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 4500 | 481 | 0 | 2873 | 153,48 | 0 | 4,996 | 36,33 | 0,87 | 7447,9 |
| Listar Asientos | 4500 | 198 | 0 | 2062 | 68,27 | 0,00022 | 5,009 | 10,35 | 1,17 | 2115,5 |
| Login | 4500 | 548 | 0 | 4836 | 267,63 | 0 | 5,008 | 2,31 | 1,45 | 473,1 |
| Registrar Reserva | 3375 | 274 | 0 | 1722 | 53,46 | 0 | 3,766 | 1,42 | 1,73 | 385,5 |
| Total | 16875 | 382 | 0 | 4836 | 220,13 | 0,00006 | 18,71 | 50,32 | 5,19 | 2753,5 |

**Figura 56.** Resultados Pruebas de 500 solicitudes

**Media Total**: 382 ms.

**Mejor Desempeño:** Listar Asientos con un tiempo promedio de 198 ms. Este endpoint muestra una baja desviación estándar (68.27 ms), lo que indica tiempos consistentes.

**Mayor Tiempo de Respuesta:** Login con un promedio de 548 ms y un máximo de 4,836 ms. Esto refleja la posible necesidad de optimización en este endpoint.

**Desviación Estándar**

- La desviación estándar total fue 220.13 ms, indicando variaciones moderadas en los tiempos de respuesta.
- Login tuvo la mayor variabilidad (267.63 ms), lo que sugiere inconsistencias bajo carga sostenida.

**Tasa de Errores** La tasa de errores fue prácticamente 0% para todos los endpoints, lo que demuestra que el sistema mantuvo la estabilidad y no rechazó solicitudes durante la carga sostenida.

**Rendimiento Total:** 18.71 solicitudes por segundo.

**Mayor Transferencia de Datos:** Listar Eventos tuvo un rendimiento de 36.33 KB/seg y envió un promedio de 7447.9 bytes por solicitud, lo que es consistente con su propósito de manejar grandes volúmenes de datos.

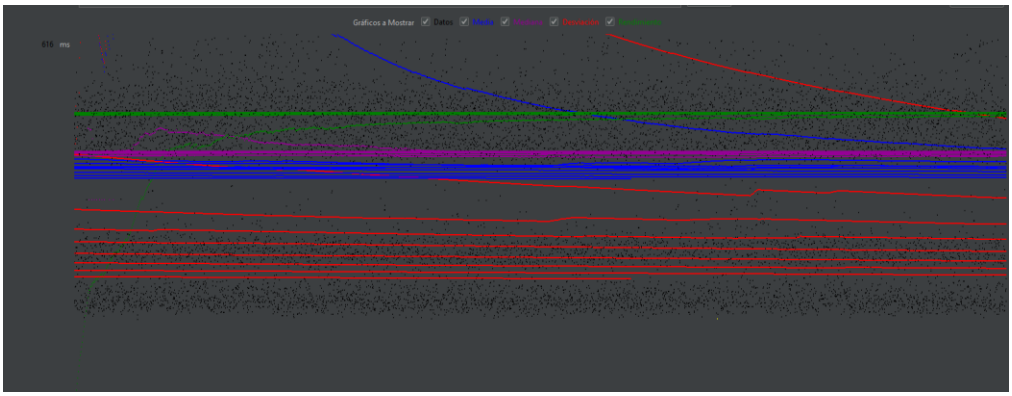


**Figura 57.** Resumen de pruebas de carga de 500 solicitudes

- 1,000 solicitudes concurrentes.

| Etiqueta | # Muestras | Media | Mediana | 90% Line | 95% Line | 99% Line | Mín | Máx | % Error | Rendimiento | Kb/sec | Sent KB |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 9000 | 436 | 425 | 460 | 481 | 655 | 382 | 2805 | 0 | 9,96543 | 72,48 | 1,73 |
| Listar Asientos | 9000 | 186 | 178 | 210 | 226 | 340 | 149 | 2041 | 0,00022 | 9,99358 | 20,62 | 2,33 |
| Login | 9000 | 592 | 487 | 519 | 547 | 4851 | 442 | 13295 | 0 | 9,99085 | 4,62 | 2,88 |
| Registrar Reserva | 6750 | 259 | 251 | 268 | 280 | 549 | 165 | 2199 | 0 | 7,51496 | 2,83 | 3,44 |
| Total | 33750 | 376 | 407 | 495 | 510 | 719 | 149 | 13295 | 0,00006 | 37,33391 | 100,37 | 10,34 |

**Figura 58.** Resultados Pruebas de 1000 solicitudes

**Media Total:** 376 ms.

**Mejor Desempeño:** Listar Asientos nuevamente sobresale con un tiempo promedio de 186 ms y un 99% de los tiempos menores a 340 ms.

**Mayor Tiempo de Respuesta:** Login sigue siendo el más lento, con un promedio de 592 ms y un máximo de 13,295 ms, indicando que podría convertirse en un cuello de botella bajo cargas altas.

**Tasa de Errores:** La tasa de errores fue casi inexistente (0.00022%) incluso bajo esta carga, lo que refleja estabilidad operativa.

**Rendimiento Total:** 37.33 solicitudes por segundo.

**Mayor Transferencia de Datos:** Listar Eventos nuevamente lidera con 72.48 KB/seg, lo que confirma su capacidad para manejar grandes volúmenes de información.

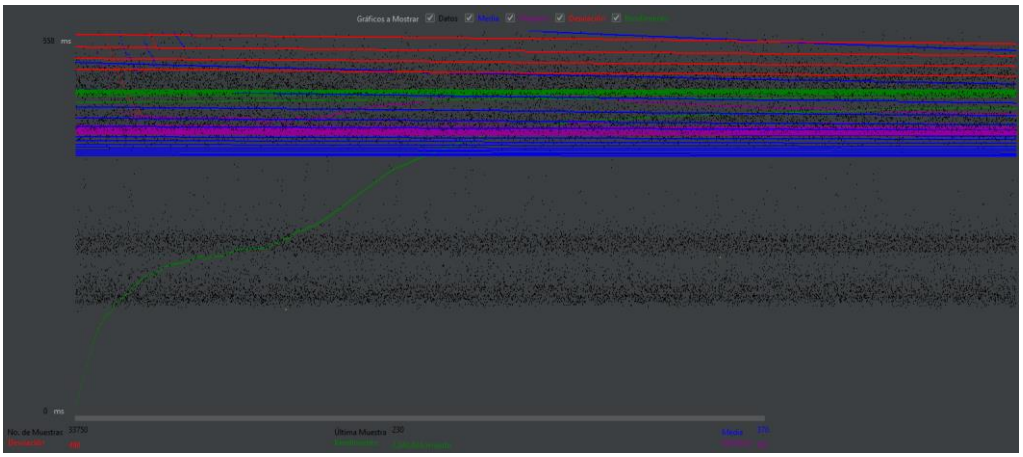


**Figura 59.** Resumen de pruebas de carga de 1000 solicitudes

### Observaciones Generales

El sistema demostró estabilidad, manteniendo una tasa de errores prácticamente nula incluso bajo 1,000 solicitudes concurrentes.

- Listar Asientos y Registrar Reserva mostraron los tiempos de respuesta más consistentes y rápidos en ambas pruebas.
- Login tuvo el peor desempeño con tiempos altos y una variabilidad significativa, lo que indica un área de mejora.

### Prueba de Picos Súbitos

- Incremento de 100 a 1,000 solicitudes en 10 segundos.

| Etiqueta | # Muestras | Media | Mín | Máx | Desv. Estándar | % Error | Rendimiento | Kb/sec | Sent KB/sec | Media de Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 1100 | 8682 | 0 | 23696 | 5371,47 | 0,02364 | 20,41763 | 145,09 | 3,55 | 7276,9 |
| Listar Asientos | 1100 | 2134 | 0 | 11833 | 1993,81 | 0,29636 | 21,24167 | 32,21 | 4,96 | 1552,9 |
| Login | 1100 | 5919 | 0 | 24457 | 4532,45 | 0,30818 | 20,58422 | 8,17 | 5,94 | 406,6 |
| Registrar Reserva | 1000 | 1839 | 0 | 13390 | 3113,24 | 0,503 | 26,93458 | 7,89 | 10,83 | 300 |
| Total | 4300 | 4709 | 0 | 24457 | 4892 | 0,27767 | 72,82951 | 173,01 | 19,77 | 2432,5 |

**Figura 60.** Resultados Pruebas de 100 a 1000 solicitudes

**Media Total:** 4,709 ms.

**Mayor Tiempo de Respuesta:** Listar Eventos: Promedio de 8,682 ms y un máximo de 23,696 ms. Este endpoint muestra tiempos elevados bajo el incremento súbito, lo que sugiere un cuello de botella en el manejo de grandes volúmenes de datos.

**Mejor Desempeño:** Registrar Reserva: Promedio de 1,839 ms, aunque su máximo llegó a 13,390 ms.

**Error Total:** 0.27%. Listar Asientos y Login mostraron tasas de error ligeramente superiores (0.296% y 0.308%, respectivamente), lo que podría estar relacionado con la saturación momentánea de recursos.

**Rendimiento Total:** 72.83 solicitudes por segundo.

**Mayor Transferencia de Datos:** Listar Eventos lidera con 145.09 KB/seg, destacándose como el endpoint más intensivo en términos de volumen.

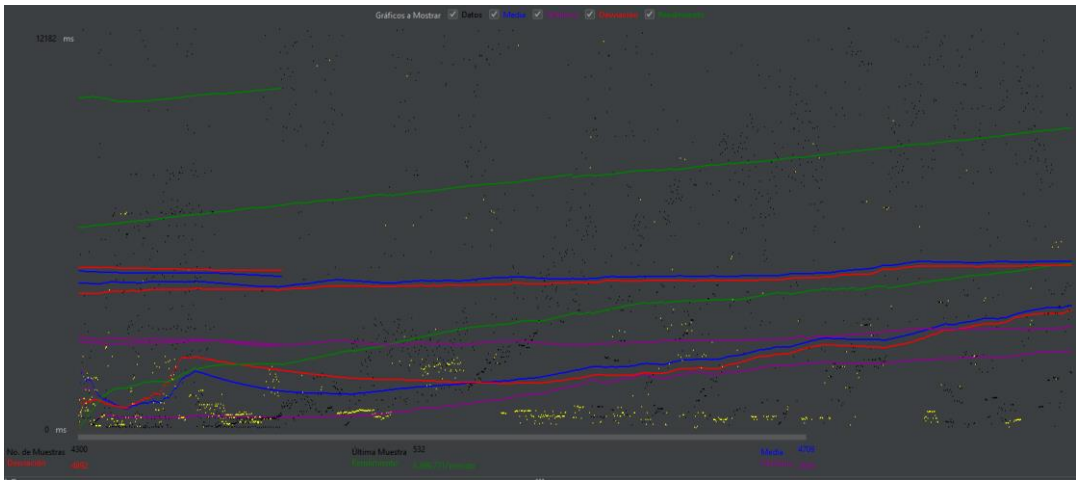


**Figura 61.** Resumen de pruebas de carga de 100 a 1000 solicitudes

- Incremento de 200 a 1,500 solicitudes en 15 segundos.

| Etiqueta | # Muestras | Media | Mín | Máx | Desv. Estándar | % Error | Rendimiento | Kb/sec | Sent KB/sec | Media de Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 2036 | 23805 | 0 | 62138 | 11766,41 | 0,08546 | 23,83768 | 164,69 | 3,81 | 7074,7 |
| Listar Asientos | 1725 | 9859 | 0 | 48614 | 11614 | 0,38551 | 20,6436 | 28,25 | 4,81 | 1401,4 |
| Login | 1265 | 2702 | 0 | 25570 | 2913,36 | 0,33518 | 18,44373 | 7,21 | 5,32 | 400 |
| Registrar Reserva | 961 | 8299 | 0 | 39741 | 13358,75 | 0,76067 | 23,77242 | 6,51 | 9,34 | 280,4 |
| Total | 5987 | 12839 | 0 | 62138 | 13596,14 | 0,33305 | 68,74813 | 197,33 | 16,88 | 2939,2 |

**Figura 62.** Resultados Pruebas de 200 a 1500 solicitudes

**Media Total:** 12,839 ms.

**Mayor Tiempo de Respuesta:** Listar Eventos: Promedio de 23,805 ms y un máximo de 62,138 ms, indicando que este endpoint es significativamente afectado por el aumento de carga.

**Registrar Reserva:** Mostró un promedio de 8,299 ms, lo que sugiere que también se ve impactado bajo cargas extremas.

**Mejor Desempeño:** Login: Aunque su tiempo promedio fue 2,702 ms, su máximo alcanzó 25,570 ms, lo que sigue siendo alto, pero más manejable comparado con otros endpoints.

**Error Total: 0.33%.** Registrar Reserva tuvo la tasa de error más alta (0.76%), lo que podría ser una señal de saturación en los recursos de base de datos o procesamiento de solicitudes concurrentes.

**Rendimiento Total:** 68.74 solicitudes por segundo.

**Mayor Transferencia de Datos:** Listar Eventos nuevamente lidera con 164.69 KB/seg, indicando una alta carga en términos de volumen de datos.

### Observaciones Generales

El sistema mantiene tasas de error bajas (menores al 1%) incluso bajo picos súbitos de carga, lo que demuestra una cierta capacidad de adaptación. Sin embargo, el tiempo de respuesta promedio y máximo aumenta significativamente, especialmente en operaciones intensivas como Listar Eventos.

- Listar Eventos es el endpoint más afectado, con tiempos de respuesta extremadamente altos bajo ambas configuraciones.
- Registrar Reserva y Login presentan tiempos manejables en promedio, pero con valores máximos preocupantes bajo cargas extremas.

## Prueba de Estrés Extremo

- 2,000 solicitudes concurrentes durante 5 minutos.

| Etiqueta | # Muestras | Media | Mín | Máx | Desv. Estándar | % Error | Rendimiento | Kb/sec | Sent KB/sec | Media de Bytes |
|---|---|---|---|---|---|---|---|---|---|---|
| Listar Eventos | 6000 | 119682 | 0 | 639908 | 166317,18 | 0,661 | 9,35578 | 37,32 | 0,81 | 4085,2 |
| Listar Asientos | 5444 | 70113 | 0 | 637851 | 108659,35 | 0,65797 | 8,50196 | 13,08 | 1,49 | 1575,5 |
| Login | 5229 | 38460 | 0 | 639172 | 60512,97 | 0,59763 | 8,17641 | 5,86 | 2,02 | 733,6 |
| Registrar Reserva | 3883 | 18381 | 0 | 615221 | 46228 | 0,72573 | 6,19921 | 4,22 | 1,97 | 696,9 |
| Total | 20556 | 66757 | 0 | 639908 | 118331,51 | 0,6563 | 32,05289 | 60,35 | 6,23 | 1927,9 |

**Figura 63.** Resultados Pruebas de 2000 solicitudes

**Media Total:** 66,757 ms (66.75 segundos).

**Peores Resultados:**

- Listar Eventos: Promedio de 119,682 ms (casi 2 minutos) con un máximo de 639,908 ms, lo que indica un cuello de botella extremo.
- Listar Asientos: Tiempo promedio de 70,113 ms con un máximo similar (637,851 ms), demostrando que ambos endpoints no escalan bien bajo esta carga.

**Mejor Desempeño:**

- Registrar Reserva: Promedio de 18,381 ms, aunque su máximo alcanzó 615,221 ms.

- Login: Aunque el promedio fue 38,460 ms, el tiempo máximo llegó a 639,172 ms, mostrando inestabilidad.

**Error Total:** 65,6%.

Registrar Reserva tuvo la tasa de error más alta (72,57%), seguida por Listar Eventos (66,1%) y Listar Asientos (65,797%).

Estos errores son una señal de saturación en los recursos de backend o base de datos.

- 3,000 solicitudes concurrentes hasta que el sistema falle.

  Este último escenario no se pudo probar dado la limitación del computador local en recursos para lanzar 3000 solicitudes concurrentes.

**Observaciones Generales**

Bajo estrés extremo, el sistema muestra tiempos de respuesta extremadamente altos y tasas de error significativas, especialmente en endpoints intensivos como Listar Eventos y Listar Asientos.

La capacidad del sistema para manejar grandes volúmenes de datos sigue siendo adecuada, aunque el rendimiento general disminuye debido a los tiempos de respuesta prolongados.

**Anexo 5.** Informe comparativo de presentación de resultados de pruebas

## Informe Comparativo de Presentación de Resultados de Pruebas

### 1. Introducción

Este informe presenta un análisis comparativo de los resultados obtenidos durante las pruebas de carga y estrés realizadas sobre los servicios **AuthCore** y **BookingCore**, implementados en **Azure Functions**. El objetivo es evaluar y contrastar su desempeño en términos de **tiempo de respuesta**, **tiempo de escalamiento automático** y **capacidad de manejo de solicitudes concurrentes**. La comparación busca identificar patrones de comportamiento bajo diferentes niveles de carga y destacar áreas de mejora.

### 2. Metodología

Las pruebas se realizaron utilizando **Apache JMeter** para generar solicitudes concurrentes y simular escenarios de carga progresiva y picos súbitos. Se recopilaron datos a través de **Azure Monitor** y **Application Insights**, enfocándose en las siguientes métricas clave:

- **Tiempo de respuesta promedio**: Rapidez con la que cada función procesa una solicitud.
- **Tiempo de escalamiento**: Duración necesaria para que Azure Functions cree nuevas instancias en respuesta a la demanda.
- **Tasa de éxito**: Porcentaje de solicitudes procesadas correctamente.
- **Número de instancias creadas**: Capacidad del sistema para escalar dinámicamente.

## 3. Resultados Generales

### 3.1 Bookincore

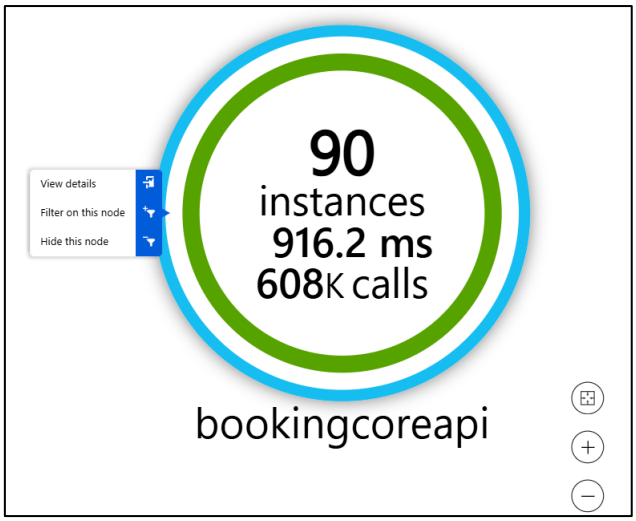



**Figura 64.** Rendimiento servicio bookingcoreapi

### 1. Análisis del Tiempo de Respuesta

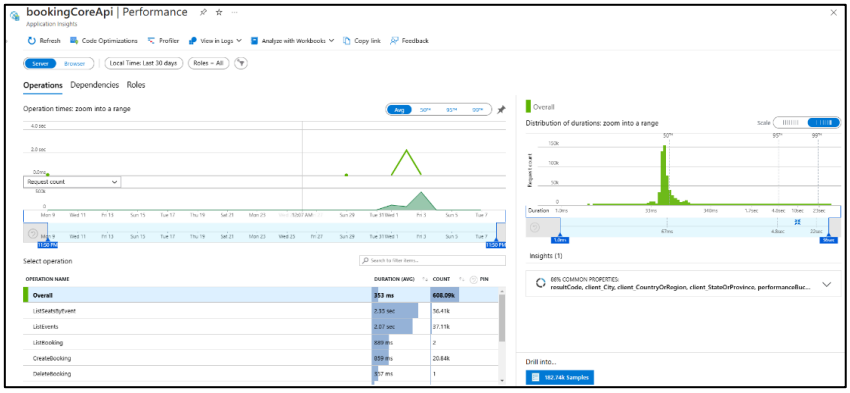



**Figura 65.** Tiempo de respuesta de servicio bookingcoreapi

### Métricas Generales

- **Duración Promedio:** El tiempo promedio de respuesta general fue de **353 ms**, aunque algunos endpoints, como ListSeatsByEvent y ListEvents, mostraron tiempos significativamente mayores (2.35s y 2.07s, respectivamente).
- **Máximo Observado:** ListEvents alcanzó un tiempo máximo de **621,138 ms**, lo cual es excesivo y señala problemas de latencia severos bajo carga alta.

### Puntos Críticos Identificados

1. **Endpoints más Lentos:**
    - ListSeatsByEvent y ListEvents tienen tiempos de respuesta promedio altos debido al acceso intensivo a la base de datos.
    - CreateBooking mostró un desempeño razonable con **859 ms**, pero esto podría empeorar bajo cargas más altas.
2. **Errores Relacionados con la Base de Datos:**
    - El error recurrente **"remaining connection slots are reserved"** indica que el sistema alcanzó el límite de conexiones permitido en la base de datos, lo que aumenta significativamente el tiempo de respuesta para solicitudes subsecuentes.

## 2. Análisis de Escalamiento

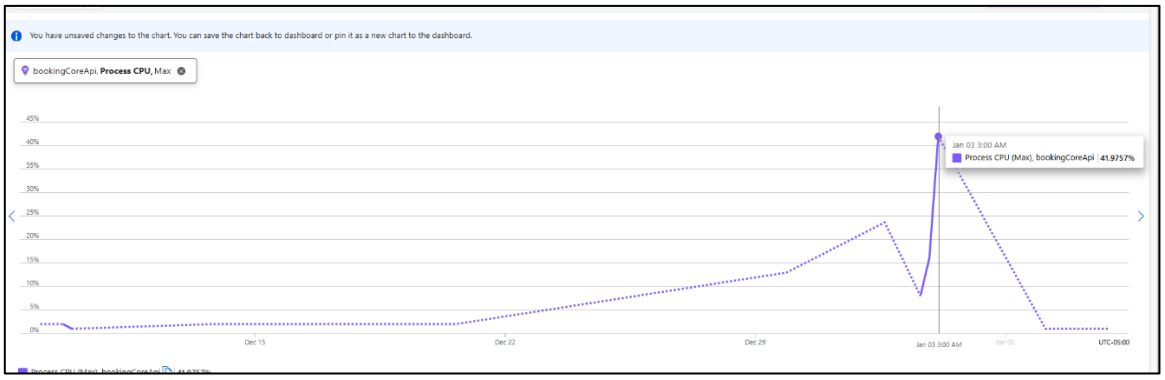



**Figura 66.** Escalamiento de servicio bookingcoreapi

### Métricas Relevantes

- **Número de Instancias:** Se crearon 90 instancias para manejar 608K solicitudes, pero el uso máximo de CPU alcanzó solo el **41%**, lo que indica que el servicio no escaló de manera efectiva.
- **Errores Críticos:** Un número elevado de errores de conexión a la base de datos impidió que las instancias adicionales manejaran la carga esperada.

**Impacto en la Escalabilidad**

1. **Limitación en la Base de Datos:**
    - La base de datos alcanzó su límite de conexiones, afectando directamente el escalamiento automático de Azure Functions.
2. **Ineficiencia del Escalamiento:**
    - Aunque se lanzaron más instancias, estas no pudieron procesar solicitudes de manera eficiente debido a la saturación de la base de datos.

**3. Errores Relevantes**

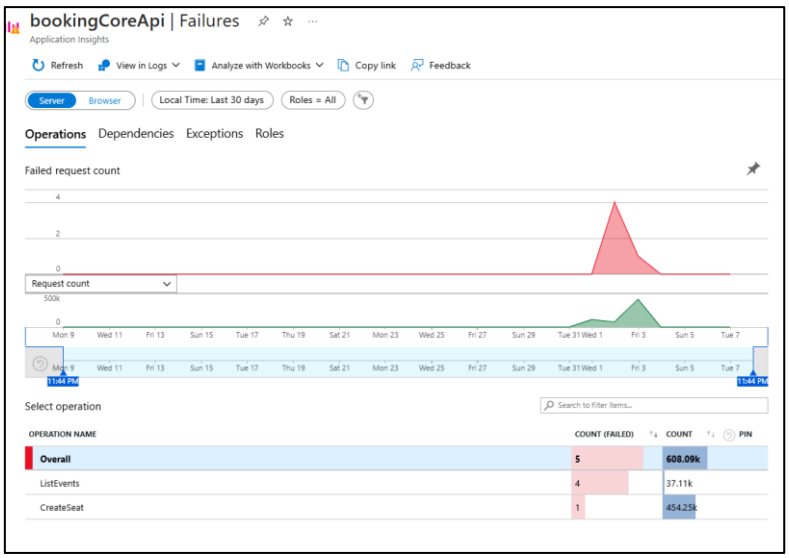



**Figura 67.** Errores servicio bookingcoreapi

**Errores Más Frecuentes:**

1. **[ERROR] "remaining connection slots are reserved":**
    - Se registraron **10,721 ocurrencias**, afectando múltiples operaciones críticas como ListEvents, ListSeatsByEvent, y CreateBooking.
    - Este problema está directamente relacionado con la configuración del límite de conexiones de PostgreSQL.
2. **[ERROR] 499:**
    - El código 499 indica solicitudes canceladas por el cliente debido a tiempos de espera prolongados.

**Impacto en las Operaciones:**

- **ListEvents y ListSeatsByEvent:**
    - Tienen los errores más frecuentes y representan puntos de falla críticos bajo alta carga.
- **CreateBooking:**
    - Aunque menos frecuente, sus errores también afectan operaciones críticas para el flujo de negocio.

### 3.2 AuthCore

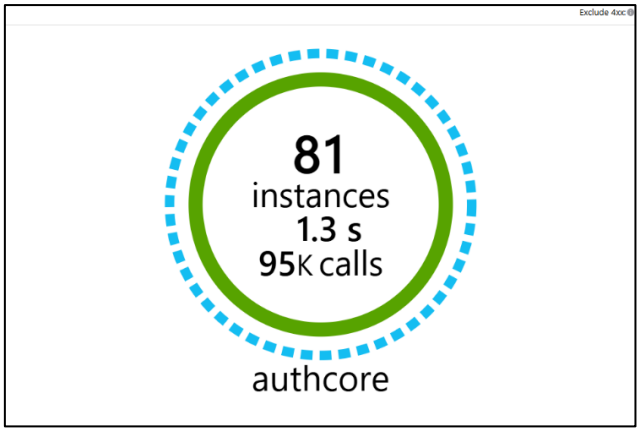



**Figura 68.** Rendimiento servicio authcore

### 1. Análisis del Tiempo de Respuesta

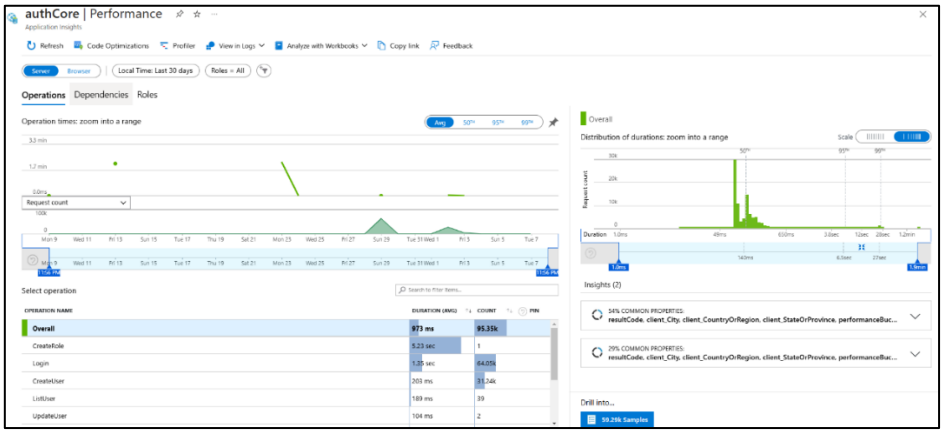



**Figura 69.** Tiempo de respuesta servicio authcore

### Métricas Generales

- **Tiempo Promedio de Respuesta General: 973 ms**, con una operación específica como Login alcanzando un tiempo promedio de **1.3 segundos**.
- **Operaciones más Lentas:**
    - Login: Tiempo promedio de **1.3 segundos**, señalando un cuello de botella en la autenticación.
    - CreateUser: **202.9 ms**, aunque razonable, su desempeño podría verse afectado en cargas más altas.
    - ListUser: **188.9 ms**, consistente, pero muestra margen de optimización.

### Errores Frecuentes Relacionados con el Tiempo de Respuesta:

- **Código de error 499**: Indica que las solicitudes fueron canceladas por el cliente debido a tiempos de espera prolongados.

### 2. Escalamiento y Utilización de Recursos

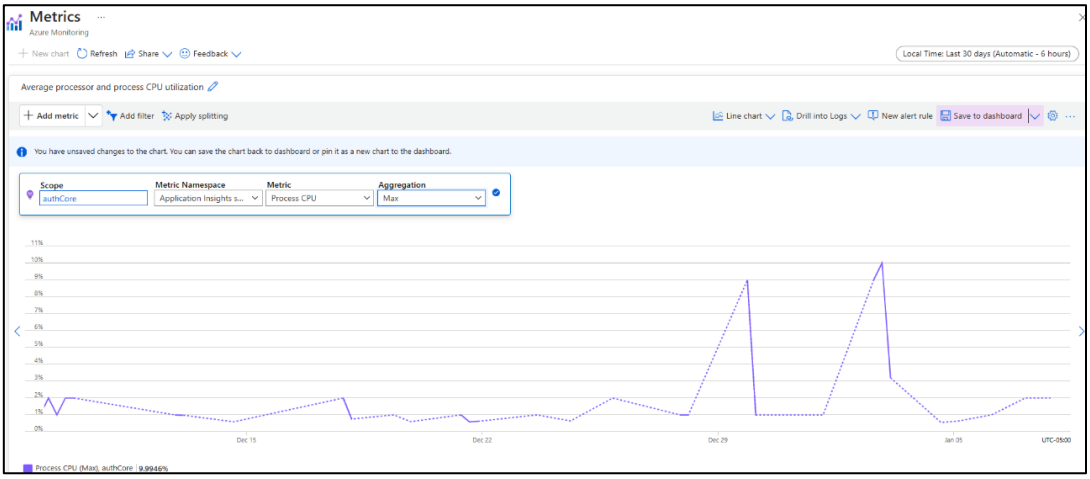



**Figura 70.** Escalamiento servicio authcore

### Métricas Relevantes

- **Número de Instancias: 81 instancias** desplegadas para manejar **95K llamadas**, lo cual sugiere una capacidad adecuada de escalamiento.
- **Uso Máximo de CPU: 9%**, muy bajo, lo que indica que el sistema no utilizó de manera eficiente los recursos disponibles para manejar la carga.

**Impacto en la Escalabilidad**

1. **Bajo Uso de Recursos:**
    - El uso máximo de CPU al 9% refleja que las instancias de Azure Functions no estuvieron limitadas por la capacidad de procesamiento.
2. **Errores de Conexión a la Base de Datos:**
    - Estos errores limitaron la capacidad de las funciones para procesar solicitudes, afectando directamente el escalamiento.

**3. Análisis de Errores**

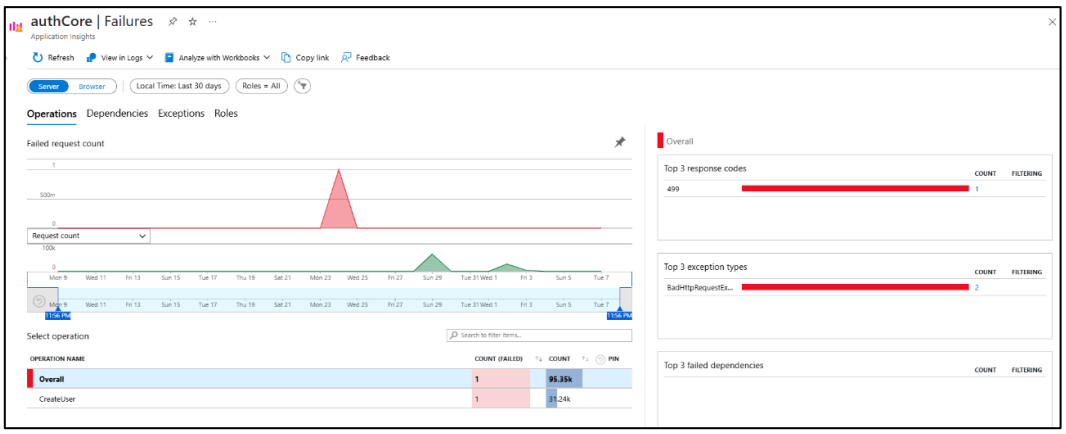



**Figura 71.** Errores servicio authcore

**Errores Frecuentes:**

1. **Conexiones Rechazadas (ECONNREFUSED):**
    - La base de datos no pudo aceptar conexiones adicionales, posiblemente debido a configuraciones insuficientes.
2. **Tiempo de Espera Excedido (ETIMEDOUT):**
    - Retrasos en la comunicación entre el servicio y la base de datos, posiblemente debido a latencia en la red o problemas de disponibilidad.

**Impacto Operativo:**

- **Login y CreateUser:**
    - Estas operaciones críticas estuvieron afectadas por errores frecuentes de conexión, lo que deteriora la experiencia del usuario.
- **Capacidad Reducida:**
    - El sistema no pudo procesar solicitudes eficientemente, llevando a tasas de errores más altas y tiempos de espera prolongados.

**Anexo 6.** Certificado de traducción del resumen de español a inglés

Loja, 10 de enero de 2025

Lic. Jessica Michelle Macas

CERTIFICO:

Que el resumen del Trabajo de titulación denominado **"Escalamiento Automático en un Sistema de Gestión de Reservas con Alta Disponibilidad Usando Azure"**, perteneciente a **Omar Alexis Sanmartin Tapia**, con cédula de identidad **1105381014**, previo a la obtención del Título de Magister en Ingeniería en Software de la Universidad Nacional de Loja; traducido al inglés cumple con las características propias del idioma extranjero.

**Resumen:**

Este trabajo de titulación aborda la implementación de un sistema de gestión de reservas utilizando una arquitectura serverless en Azure Functions para garantizar alta disponibilidad y escalabilidad. Inicialmente, el sistema fue desarrollado en un modelo basado en servidor utilizando Express.js, pero debido a los objetivos del proyecto, se migró a una arquitectura serverless, implicando un refactor significativo del código para adaptarlo a las características de este entorno.

El proyecto incluyó la configuración de servicios serverless, la integración con sistemas de escalamiento automático y la ejecución de pruebas de carga y estrés para evaluar su comportamiento bajo distintas condiciones. Durante las pruebas, se observó que los servicios no alcanzaron un escalamiento efectivo, con un uso máximo de CPU de 9% en AuthCore y 41% en BookingCore. Adicionalmente, se identificaron cuellos de botella en la base de datos, que resultaron en errores frecuentes de conexión y tiempos de respuesta elevados en escenarios de alta carga.

Este trabajo concluye que, si bien las arquitecturas serverless ofrecen una base prometedora para aplicaciones críticas, su desempeño depende directamente de la configuración y optimización de la infraestructura de soporte, como el manejo de conexiones de base de datos y el uso de mecanismos de caché para reducir la dependencia de operaciones intensivas en recursos.

**Abstract:**

This degree work addresses the implementation of a reservation management system using a serverless architecture in Azure Functions to ensure high availability and scalability. Initially, the system was developed in a server-based model using Express.js, but due to the objectives of the project, it was migrated to a serverless architecture, involving a significant refactoring of the code to adapt it to the characteristics of this environment.

The project included the configuration of serverless services, the integration with automatic scaling systems and the execution of load and stress tests to evaluate its behavior under different conditions. During the tests, it was observed that the services did not scale effectively, with a maximum CPU usage of 9% in AuthCore and 41% in BookingCore. Additionally, database bottlenecks were identified, resulting in frequent connection errors and high response times in high load scenarios.

This paper concludes that while serverless architectures offer a promising foundation for critical applications, their performance is directly dependent on the configuration and optimization of the supporting infrastructure, such as database connection handling and the use of caching mechanisms to reduce dependency on resource-intensive operations.

Lic. Jessica Michelle Macas