Arquitectura de Servicios para la Gestión de Garantías mediante Notificaciones Calendarizadas

Plan de Pruebas

Versión <1.1.0>

Historia de Revisiones

|  |  |  |  |
| --- | --- | --- | --- |
| **Fecha** | **Versión** | **Descripción** | **Autor** |
| 2025-06-20 | 1.1.0 | Documento inicial | Renzo Gomez |
| 2025-07-06 | 1.1.0 | Ajustes en la documentación | Renzo Gomez |
| 2025-07-09 | 1.1.0 | Refinación final documentación | Renzo Gomez |
|  |  |  |  |

Tabla de Contenidos

1 Introducción 4

1.1 Propósito 4

1.2 Ámbito 4

1.2.1 Pruebas unitarias 4

1.2.2 Pruebas funcionales 4

1.2.3 Riesgos 4

1.2.4 Identificación del proyecto 6

1.3 Breve Descripción 6

1.4 Definiciones, acrónimos y abreviaciones 6

1.5 Referencias 7

2 Requerimientos para pruebas 8

2.1 Casos de uso 8

2.1.1 Vista global 8

2.1.2 Caso de uso 1 8

2.1.3 Caso de uso 2 8

2.1.4 Caso de uso 3 8

2.2 Requerimientos funcionales 8

2.2.1 Componentes comunes 8

2.2.2 Componente 1 8

2.2.3 Componente 2 8

2.3 Requerimientos No-Funcionales (Matriz con Input/Output, solo si aplica) 8

2.3.1 Componente 1 (puede tener uno o más casos de uso) 8

2.3.2 Componente 2 (puede tener uno o más casos de uso) 8

3 Estrategia de Pruebas 9

3.1 Tipos de pruebas 9

3.1.1 Integridad de la base de datos y de los datos 9

3.1.2 Pruebas funcionales 10

3.1.3 Pruebas de rendimiento (Performance) 11

3.1.4 Pruebas de seguridad y de acceso a datos 11

3.1.5 Herramientas involucradas 12

4 Recursos 13

4.1 Profesionales 13

4.2 Ambiente de pruebas 14

4.2.1 Preparación del ambiente de pruebas 14

4.2.2 Diseño del ambiente de pruebas 14

4.2.3 Diseño ambiente de pruebas 16

4.2.4 Integración del ambiente de pruebas y configuración 17

4.2.5 Definición del banco de datos de prueba 17

4.2.6 Generación de datos 17

5 Actividades e Hitos del Plan de Pruebas 20

6 Entregables 22

6.1 Plan de pruebas 22

6.1.1 Criterio de entrada para el “Plan de pruebas” 22

6.1.2 Criterio de salida para el “Plan de pruebas” 22

6.1.3 Criterio de suspensión y resumisión 22

6.2 Resultados de las pruebas 22

6.3 Reporte de defectos 22

7 Anexos 23

7.1 A: Tareas del proyecto 23

7.2 B: Pruebas de rendimiento (*performance*) 24

7.3 C: Pruebas de seguridad y de control de acceso 25

# Introducción

## Propósito

El propósito de este documento es definir el plan de validación técnica de la **arquitectura escalable basada en servicios**, diseñada y desplegada mediante herramientas de **infraestructura como código (IaC)**, específicamente **OpenTofu**. Este plan establece las pruebas que se realizarán para verificar que la infraestructura y los componentes arquitectónicos cumplen con los requisitos técnicos establecidos en cuanto a **desempeño, escalabilidad, disponibilidad y automatización**.

A diferencia de un enfoque centrado únicamente en la funcionalidad de una aplicación específica, este documento se orienta a validar el correcto funcionamiento de los elementos que conforman la arquitectura —como servicios desacoplados, almacenamiento de datos, canales de comunicación y automatización del despliegue— asegurando que el sistema sea confiable, reproducible y adaptable a escenarios futuros de mayor complejidad.

Este plan servirá como base para garantizar que las decisiones de diseño adoptadas, así como las herramientas seleccionadas, son apropiadas para sostener soluciones como *Recooba* u otras plataformas que se construyan sobre esta misma arquitectura.

## Ámbito

Este Plan de Pruebas está asociado al proyecto de diseño y despliegue de una **arquitectura de servicios escalable**, automatizada mediante herramientas de **infraestructura como código (IaC)**, específicamente **OpenTofu**. Si bien esta arquitectura puede ser utilizada por aplicaciones como *Recooba*, su validación se centra en los componentes estructurales y técnicos que la sostienen, más allá de una implementación puntual.

El alcance de este documento incluye:

* Validación del **proceso de despliegue automático** de la infraestructura utilizando OpenTofu en entornos de desarrollo.
* **Verificación del comportamiento de los servicios básicos** (API Gateway, Backend, Base de Datos, almacenamiento) tras el aprovisionamiento.
* **Pruebas de tolerancia a fallos**, incluyendo recuperación tras caída de servicios y persistencia de datos.
* **Evaluación de la escalabilidad técnica** simulada mediante múltiples instancias o carga concurrente.
* Validación de las **tácticas arquitectónicas implementadas** (modularidad, desacoplamiento, logs, separación de capas).
* Pruebas de consistencia y disponibilidad de servicios a través de endpoints expuestos vía API.

Quedan excluidas de este alcance:

* Las pruebas funcionales propias de una aplicación específica (como flujos de usuario en *Recooba*), que serán abordadas en documentos independientes.
* Pruebas sobre sistemas operativos o dispositivos finales (Android/iOS).
* Integraciones externas no incluidas en el entorno actual (como balanceadores, redes privadas, CDN o sistemas de autenticación empresarial).
* Pruebas de interfaz o usabilidad.

### Pruebas unitarias

Las pruebas unitarias en este contexto están orientadas a la **validación de módulos técnicos independientes** dentro del ecosistema de la arquitectura. Estas pruebas no se enfocan en funciones de usuario final, sino en comprobar que cada componente de la arquitectura —ya sea un servicio, script de provisión, función de backend o recurso de infraestructura— se comporta correctamente de forma aislada.

En particular, se realizarán pruebas unitarias sobre:

* **Módulos del backend**: funciones de autenticación, controladores REST, lógica de validación y operaciones sobre la base de datos.
* **Scripts de aprovisionamiento (IaC)**: testeo de sintaxis, estructura y correcta interpretación por parte del motor OpenTofu.
* **Funciones auxiliares reutilizables**: como formateadores de logs, validadores de entrada y handlers de errores.

Las herramientas de prueba consideradas incluyen:

* **Jest o Mocha/Chai** para funciones del backend (Node.js).
* Validación con tofu validate y tofu plan para asegurar que los recursos declarados sean consistentes, completos y sin conflictos.
* Linting de código para mantener buenas prácticas en los módulos reutilizables.

Estas pruebas se ejecutarán de forma local durante el desarrollo, como primer nivel de verificación antes de avanzar a pruebas de integración o despliegue.

### Pruebas funcionales

Las pruebas funcionales en el contexto de esta arquitectura no se enfocan en la experiencia del usuario final o en pantallas específicas, sino en verificar que **los servicios desplegados cumplen correctamente las funciones que les fueron asignadas**, operando como unidades lógicas de negocio dentro de la infraestructura.

Estas pruebas se realizarán una vez completado el despliegue automatizado mediante OpenTofu, y tienen como objetivo confirmar que:

* Las **API expuestas** funcionan correctamente y responden con el comportamiento esperado ante solicitudes válidas e inválidas.
* La lógica de negocio del backend (por ejemplo, creación, lectura y eliminación de datos) se ejecuta sin errores.
* El sistema **responde de forma coherente ante distintos flujos de operación**, incluso cuando los servicios están desacoplados o distribuidos.
* Las **reglas técnicas** configuradas en el entorno (rutas, puertos, variables de entorno, seguridad básica) están correctamente aplicadas.

Estas pruebas se realizarán manualmente y con herramientas como **Postman**, **cURL**, o scripts automatizados simples que simulen solicitudes a los endpoints. A futuro, estas validaciones podrán integrarse a pipelines de CI/CD para pruebas funcionales automatizadas.

### Riesgos

Durante el proceso de validación de la arquitectura escalable mediante infraestructura como código, pueden surgir distintos riesgos técnicos y operativos que es importante identificar para tomar medidas preventivas. A continuación se detallan los principales riesgos asociados:

|  |  |  |  |
| --- | --- | --- | --- |
| **Riesgo** | **Descripción** | **Posible impacto** | **Mitigación** |
| **Errores en el aprovisionamiento con OpenTofu** | Configuraciones mal definidas en los archivos .tf pueden provocar fallos en el despliegue o inconsistencias entre entornos. | Interrupción del proceso de pruebas, pérdida de tiempo o recursos. | Validación previa con tofu validate y ejecución controlada de tofu plan. Uso de módulos reutilizables probados. |
| **Falta de trazabilidad en el registro de errores** | Si los servicios desplegados no incluyen logs estructurados, es difícil diagnosticar fallas. | Dificultad para resolver problemas, mayor tiempo de diagnóstico. | Incorporar mecanismos de logging centralizado desde el diseño. |
| **Servicios no disponibles tras despliegue** | Fallos en la conectividad entre componentes (por ejemplo, API y base de datos) pueden invalidar la arquitectura desplegada. | Fallas funcionales, pruebas bloqueadas. | Verificación de variables de entorno, puertos y dependencias tras cada despliegue. |
| **Ambiente de pruebas insuficiente** | Recursos limitados (RAM, CPU, red) pueden afectar la validez de las pruebas de carga o rendimiento. | Resultados no representativos, degradación no esperada. | Realizar pruebas controladas y escalar el entorno si es necesario para simulaciones realistas. |
| **Falta de automatización futura** | Al no integrar las pruebas en pipelines CI/CD, se puede perder continuidad en la validación posterior. | Reducción de eficiencia, errores repetitivos. | Diseñar una arquitectura compatible con integración continua desde el inicio. |

Estos riesgos serán monitoreados durante el desarrollo y validación de la arquitectura, y documentados para futuras iteraciones o escalamientos hacia ambientes productivos.

#### **Riesgos identificados:**

Durante el diseño, montaje y validación de una arquitectura escalable basada en servicios, se identifican diversos riesgos técnicos y operativos que pueden impactar la calidad, estabilidad y reproducibilidad del sistema. Si bien herramientas como **OpenTofu** permiten automatizar y estandarizar el despliegue, los riesgos están asociados al funcionamiento **global de la arquitectura** más que a una sola tecnología.

Los principales riesgos identificados son:

1. **Errores en la definición de componentes arquitectónicos**  
   Diseños inadecuados en la separación de capas, dependencia excesiva entre módulos o ausencia de componentes críticos (como gateways, autenticadores, balanceadores) pueden comprometer la escalabilidad y el mantenimiento del sistema.
2. **Incompatibilidad entre servicios**  
   La arquitectura basada en componentes desacoplados requiere contratos de integración bien definidos. Errores en endpoints, puertos, variables de entorno o estructuras de datos pueden romper el flujo de operación entre módulos.
3. **Falta de automatización y versionado de infraestructura**  
   Sin herramientas de infraestructura como código (IaC), los entornos pueden volverse inconsistentes, frágiles y difíciles de replicar. OpenTofu actúa aquí como una solución, pero el riesgo existe si no se gestiona correctamente.
4. **Despliegue en ambientes limitados o poco realistas**  
   Probar la arquitectura en entornos con recursos restringidos (CPU, RAM, red) puede ocultar problemas que surgirían en condiciones reales de producción o crecimiento.
5. **Baja observabilidad y monitoreo**  
   La ausencia de herramientas de logs centralizados, métricas o alertas puede dificultar la identificación de cuellos de botella, degradaciones o errores silenciosos.
6. **Ausencia de automatización en validación**  
   Validar manualmente cada instancia de la arquitectura introduce riesgos de error humano, baja trazabilidad y lentitud para detectar fallos.
7. **Falta de control de cambios estructurales**  
   Cambios en la arquitectura sin un sistema formal de control de versiones o documentación pueden generar inconsistencias entre equipos o ambientes.

#### Matrices de riesgos

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **ID** | **Riesgo** | **Probabilidad** | **Impacto** | **Nivel de Riesgo** | **Acción de Mitigación** |
| R1 | Errores en la definición de componentes arquitectónicos | Media | Alta | Alto | Validar la arquitectura mediante revisiones técnicas y diagramas. |
| R2 | Incompatibilidad entre servicios (API, variables, puertos) | Alta | Alta | Crítico | Pruebas de integración tempranas y contratos bien definidos. |
| R3 | Falta de automatización de infraestructura | Media | Alta | Alto | Aplicar IaC desde el inicio con versionado y pruebas (plan, apply). |
| R4 | Pruebas en entornos poco representativos | Alta | Media | Alto | Simular condiciones de carga y escalar infraestructura temporalmente. |
| R5 | Falta de observabilidad (logs y monitoreo) | Alta | Media | Alto | Incorporar logs estructurados y herramientas de monitoreo básico. |
| R6 | Validación manual sin automatización | Media | Media | Medio | Incorporar pruebas básicas automatizadas en futuras iteraciones. |
| R7 | Cambios no versionados en arquitectura o IaC | Media | Alta | Alto | Uso de repositorios Git, control de versiones y documentación técnica. |

##### Pruebas

|  |  |  |  |
| --- | --- | --- | --- |
| **Tipo de Prueba** | **Riesgo Asociado** | **Componente Afectado** | **Objetivo de Validación** |
| Pruebas Unitarias | R1, R6 | Funciones backend, scripts IaC | Verificar funcionamiento correcto y aislado de módulos. |
| Pruebas de Integración | R2, R3, R4 | Comunicación entre servicios | Comprobar interacción entre componentes y flujos de datos consistentes. |
| Pruebas de Despliegue | R3, R4, R7 | Infraestructura completa (IaC) | Asegurar que el entorno se configura correctamente en cada ejecución. |
| Pruebas de Recuperación | R2, R5 | Servicios backend y base de datos | Validar tolerancia a fallos y capacidad de recuperación. |
| Pruebas Funcionales | R2 | Endpoints de API y lógica de negocio | Confirmar que los servicios expuestos operan correctamente. |
| Pruebas de Observabilidad | R5 | Logs, métricas, trazas | Evaluar la visibilidad del sistema ante errores y eventos. |

##### Proyecto: Arquitectura de Servicios Escalable con Infraestructura como Código

| **Nº** | **Descripción** | **Gravedad** |
| --- | --- | --- |
| 1 | Fallos en la conexión entre servicios (API Gateway Base de Datos) | Medio |
| 2 | Errores en validación de entradas desde frontend | Bajo |
| 3 | Pérdida de datos por eliminación accidental sin respaldo | Alto |
| 4 | Logs de auditoría que no se registran adecuadamente | Medio |
| 5 | Inconsistencias visuales en clientes móviles de diferentes tamaños | Bajo |
| 6 | Fallos en el control de autenticación y autorización de usuarios | Alto |

### Identificación del proyecto

El Plan de Pruebas del proyecto **Recooba** está relacionado con los siguientes documentos y artefactos generados a lo largo del ciclo de desarrollo del sistema. Esta identificación permite garantizar coherencia entre las pruebas planificadas y los entregables del proyecto.

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Documento  (versión / fecha)** | **Creado o Disponible** | **Recibido o Revisado** | **Autor u Origen** | **Notas** |
| Diseño de la Arquitectura Escalable | ◼Si 🞏 No | ◼ Si 🞏 No | Renzo Gómez | Contiene visión general, modularidad y componentes desplegables |
| Infraestructura como Código (OpenTofu) | ◼ Si 🞏 No | ◼ Si 🞏 No | Renzo Gómez | Scripts de provisión para ambientes reproducibles |
| Plan de Despliegue | ◼ Si 🞏 No | ◼ Si 🞏 No | Renzo Gómez | Despliegue en VPS/Render y entorno de pruebas simulado |
| Modelos de Diseño | ◼ Si 🞏 No | ◼ Si 🞏 No | Renzo Gómez | Diagramas C4, componentes, conectores y vista de procesos |
| Matriz de Pruebas de Arquitectura | ◼ Si 🞏 No | ◼ Si 🞏 No | Renzo Gómez | Define qué se prueba en cada capa de la arquitectura |
| Aplicación de Validación (Recooba) | ◼ Si 🞏 No | ◼ Si 🞏 No | Renzo Gómez | Aplicación móvil conectada a la arquitectura como entorno demostrativo |
| Manual de Pruebas | ◼ Si 🞏 No | 🞏 Si ◼ No | Renzo Gómez | Procedimientos de validación y criterios de aceptación |

## Breve Descripción

Recooba es una aplicación móvil desarrollada en React Native, utilizada en este proyecto como **caso de prueba para validar el despliegue y funcionamiento de una arquitectura de servicios escalable**. La aplicación permite simular flujos reales de uso sobre una infraestructura basada en microservicios, base de datos relacional, y control de versiones.

A través de Recooba se evalúan aspectos funcionales como autenticación, registro de datos, consultas, y notificaciones, los cuales permiten observar cómo la arquitectura responde ante operaciones CRUD, persistencia de información, seguridad, y trazabilidad.

Esta aplicación se conecta a un backend desarrollado en Node.js y Express.js, con una base de datos MariaDB, desplegado mediante infraestructura como código utilizando OpenTofu.

#### Funcionalidades principales (Aplicación de validacion):

Recooba implementa las siguientes funciones principales, que permiten validar la integridad y eficiencia de la arquitectura de servicios:

* **Registro de garantías** con campos como: nombre del producto, tienda, duración, fecha de compra y fotografía de la boleta.
* **Autenticación de usuarios** mediante correo electrónico y contraseña.
* **Consulta de garantías registradas**, visualizadas en una interfaz ordenada y amigable.
* **Edición y eliminación de garantías** desde la base de datos mediante peticiones API.
* **Notificaciones automáticas** antes del vencimiento de las garantías (mediante sistema de schedulers).

La información es almacenada en una base de datos relacional **MariaDB**, y las operaciones de backend se gestionan mediante servicios REST. El sistema se ejecuta sobre dispositivos Android, y se proyecta su escalabilidad hacia servicios externos como respaldo en la nube, exportación de información, y automatización mediante OCR o IA.

## Definiciones, acrónimos y abreviaciones

|  |  |
| --- | --- |
| **Término / Acrónimo** | **Definición** |
| API | Application Programming Interface. Conjunto de reglas que permite la comunicación entre sistemas o servicios. |
| Backend | Parte del sistema que se encarga de la lógica de negocio, acceso a datos y procesamiento de solicitudes. |
| Frontend | Capa de presentación de una aplicación con la que interactúa el usuario. |
| Recooba | Aplicación móvil utilizada como entorno de validación para la arquitectura de servicios escalable. |
| MariaDB | Sistema de gestión de bases de datos relacional utilizado para la persistencia de datos. |
| Node.js | Entorno de ejecución para JavaScript en el lado del servidor. |
| React Native | Framework para el desarrollo de aplicaciones móviles multiplataforma. |
| AsyncStorage | Mecanismo de almacenamiento local en React Native para guardar datos clave-valor. |
| OpenTofu | Herramienta de IaC (Infraestructura como Código) utilizada para desplegar componentes de la arquitectura. |
| ORM | Object-Relational Mapping. Técnica para mapear objetos de programación a tablas de bases de datos. |
| JWT | JSON Web Token. Estándar para transmitir información segura entre partes como un objeto JSON. |
| HTTP | Hypertext Transfer Protocol. Protocolo de comunicación utilizado en la web. |
| CRUD | Operaciones básicas de persistencia: Crear, Leer, Actualizar y Eliminar. |
| Logs | Registros de eventos o acciones del sistema, útiles para auditoría y detección de errores. |
| Arquitectura Modular | Modelo de diseño que divide el sistema en componentes reutilizables e independientes. |
| Escalabilidad | Capacidad de un sistema para manejar mayor carga de trabajo sin afectar su rendimiento. |

## Referencias

1. OpenTofu (2024). Infrastructure as Code with OpenTofu Documentation. Recuperado de: https://opentofu.org/docs
2. React Native (2024). React Native Documentation. Meta Platforms Inc. Disponible en: <https://reactnative.dev>
3. MariaDB Foundation (2024). MariaDB Server Documentation. Recuperado de: https://mariadb.com/kb/en/
4. Node.js (2024). Node.js v20 Documentation. Node.js Foundation. Recuperado de: https://nodejs.org/en/docs
5. ISO/IEC/IEEE 29119-1:2013. Software and systems engineering — Software testing — Part 1: Concepts and definitions.
6. IEEE 1012-2016. Standard for System and Software Verification and Validation.
7. Sommerville, I. (2011). Ingeniería del software (9a ed.). Pearson Educación.
8. Pressman, R. S., & Maxim, B. R. (2015). Software Engineering: A Practitioner's Approach (8th ed.). McGraw-Hill.
9. Documentación técnica del proyecto:
   * Documento de Arquitectura del Sistema (DAS)
   * Modelo de Casos de Uso
   * Plan de Despliegue
   * Matriz de Riesgos
   * Manual de Validación

## Organización del documento

Este documento está estructurado de manera que permita comprender, planificar y ejecutar pruebas enfocadas en validar una arquitectura de servicios escalable basada en principios de modularidad, automatización e infraestructura como código.

El contenido se distribuye de la siguiente forma:

* **Sección 1 – Introducción:** Define el propósito, alcance, enfoque y contexto del proyecto. Se describen los elementos claves utilizados, como OpenTofu, Recooba como entorno de validación, y los principios arquitectónicos considerados.
* **Sección 2 – Planificación de pruebas:** Establece el cronograma, recursos, responsables y actividades planificadas para ejecutar las pruebas sobre la arquitectura desplegada.
* **Sección 3 – Casos de prueba:** Detalla los escenarios específicos diseñados para validar cada componente clave del sistema, tanto funcionales como no funcionales (desempeño, seguridad, escalabilidad).
* **Sección 4 – Criterios de aceptación:** Define los criterios que deben cumplirse para que la arquitectura sea considerada válida en términos técnicos y de negocio.
* **Sección 5 – Resultados esperados y métricas:** Presenta las expectativas asociadas a cada tipo de prueba, junto con las métricas utilizadas para evaluar el comportamiento del sistema.
* **Sección 6 – Consideraciones finales:** Resume los hallazgos, aprendizajes, limitaciones y recomendaciones para implementaciones futuras.

Esta estructura permite evaluar la arquitectura desde un enfoque técnico, validando que los componentes desplegados funcionan correctamente, son reutilizables y escalables para distintas implementaciones.

# Requerimientos para pruebas

## Esta sección detalla los recursos necesarios para ejecutar las pruebas que permitan validar técnica y funcionalmente la arquitectura de servicios escalable diseñada. La infraestructura será gestionada mediante herramientas de provisión automática (OpenTofu), permitiendo la creación y destrucción de ambientes de prueba de forma reproducible.Casos de uso

#### 2.1.1 Infraestructura Base

* **Proveedor:** Plataforma de pruebas local (ambiente de desarrollo) o VPS en Render/Hostinger.
* **Recursos mínimos:**
  + CPU: 2 núcleos
  + RAM: 2 GB
  + Almacenamiento: 10 GB
  + Sistema Operativo: Linux Ubuntu 22.04 o superior
* **Accesos habilitados:** Puertos 22 (SSH), 80/443 (HTTP/HTTPS), 3306 (MariaDB), 3000–4000 (backend API, servicios auxiliares).

#### 2.1.2 Servicios Desplegados

* Contenedor o instancia Node.js para backend
* Contenedor o servicio MariaDB
* API Gateway local o vía NGINX
* Logs locales o en archivos del sistema
* OpenTofu para aprovisionamiento y pruebas de escalado/despliegue

#### 2.1.3 Herramientas para Validación

* **Postman** o **cURL** para validar endpoints
* **Visual Studio Code** con extensiones para Terraform/OpenTofu
* **DBeaver** o similar para inspección de base de datos
* **Git** para control de versiones
* **Docker** (opcional) para emulación local de servicios

### **Requerimientos Humanos (Opcional)**

* Perfil DevOps Junior o desarrollador full-stack
* Conocimiento básico en lectura de logs, interpretación de errores, despliegue de servicios

## Requerimientos funcionales

Los siguientes requerimientos funcionales corresponden a las capacidades mínimas que debe cumplir la arquitectura de servicios escalable diseñada, con el objetivo de soportar correctamente aplicaciones como Recooba u otras soluciones futuras.

|  |  |
| --- | --- |
| **Código** | **Requerimiento funcional** |
| RF-01 | La arquitectura debe permitir el **registro de usuarios** a través de servicios expuestos mediante API REST. |
| RF-02 | La arquitectura debe permitir **autenticación de usuarios** mediante tokens JWT con validación segura. |
| RF-03 | Debe ser posible **persistir, consultar, actualizar y eliminar información** en una base de datos relacional (MariaDB). |
| RF-04 | El sistema debe exponer servicios de backend que respondan a solicitudes HTTP desde un frontend externo. |
| RF-05 | Debe ser posible **ejecutar servicios asincrónicos** como envío de notificaciones programadas. |
| RF-06 | La arquitectura debe ser capaz de gestionar la **trazabilidad de operaciones** mediante logs almacenados local o remotamente. |
| RF-07 | La arquitectura debe permitir su **provisión automática y reproducible** mediante herramientas de IaC como OpenTofu. |
| RF-08 | Debe poder escalar horizontalmente los servicios según demanda (simulado en entorno de prueba). |
| RF-09 | El sistema debe permitir la **conexión concurrente de múltiples usuarios** sin pérdida de integridad de datos. |
| RF-10 | La arquitectura debe permitir **la separación de capas** (presentación, lógica de negocio y persistencia). |

## Requerimientos No-Funcionales (Matriz con Input/Output, solo si aplica)

|  |  |
| --- | --- |
| **Código** | **Requerimiento no funcional** |
| RNF-01 | El sistema debe responder a las solicitudes de API REST en un tiempo máximo de 2 segundos en condiciones normales. |
| RNF-02 | La arquitectura debe poder ser desplegada de forma **automática y repetible** utilizando OpenTofu. |
| RNF-03 | Debe permitir el monitoreo de recursos y servicios mediante logs y/o herramientas externas (opcional). |
| RNF-04 | La arquitectura debe ser **modular**, permitiendo el reemplazo de componentes sin afectar al sistema completo. |
| RNF-05 | Debe asegurar **disponibilidad mínima del 95%** durante las pruebas de estabilidad. |
| RNF-06 | Debe soportar al menos 10 usuarios concurrentes simulados sin caída de servicios. |
| RNF-07 | Toda comunicación entre frontend y backend debe realizarse mediante **protocolos seguros (HTTPS)**. |
| RNF-08 | La arquitectura debe ser fácilmente **portable** entre entornos de prueba y producción. |
| RNF-09 | Se debe poder **escalar horizontalmente** servicios específicos (por ejemplo, backend) de forma controlada. |
| RNF-10 | Los datos deben mantenerse **consistentes e íntegros**, incluso ante errores o caídas controladas de componentes. |

# Estrategia de Pruebas

La estrategia de pruebas para este proyecto se enfoca en la validación de una arquitectura de servicios escalable construida mediante principios de infraestructura como código (IaC), utilizando OpenTofu como herramienta principal de aprovisionamiento.

El objetivo es asegurar que los servicios desplegados respondan correctamente, se comuniquen entre sí de manera segura, sean escalables y cumplan con los requerimientos definidos en las secciones anteriores.

## Tipos de pruebas

### Integridad de la base de datos y de los datos

La validación de esta prueba se centra en asegurar que el aprovisionamiento de la capa de persistencia mediante infraestructura como código (IaC) con OpenTofu genere una instancia de base de datos (MariaDB) completamente funcional, estructurada, y capaz de mantener la integridad de los datos.

Se evaluará que las relaciones entre tablas, restricciones, y estructuras definidas en los scripts de despliegue se mantengan consistentes luego del proceso de aprovisionamiento, y que la base de datos permita ejecutar operaciones CRUD sin pérdida ni corrupción de información.

El propósito de esta validación es comprobar que la base de datos desplegada forma parte de un entorno de arquitectura escalable y replicable, tal como se espera en entornos reales de producción.

|  |  |
| --- | --- |
| **Objetivo de la prueba:** | **Validar que la arquitectura desplegada mediante OpenTofu sea capaz de aprovisionar correctamente servicios de base de datos (MariaDB), garantizando integridad referencial y consistencia en operaciones básicas (CRUD).** |
| **Técnica a utilizar:** | - Ejecución de scripts automatizados para operaciones INSERT, SELECT, UPDATE, DELETE sobre estructuras reales. - Verificación con herramientas como DBeaver o scripts SQL. - Simulación de fallos en la capa de datos para observar comportamiento ante errores. |
| **Criterio de validación:** | - Las operaciones deben completarse sin errores. - No deben existir registros huérfanos, duplicados ni datos incoherentes. - La estructura debe mantenerse estable tras múltiples ciclos de ejecución. |
| **Consideraciones especiales:** | - Las pruebas se ejecutan en un entorno de pruebas desplegado automáticamente vía OpenTofu. - Se sugiere monitoreo con herramientas de observabilidad (como logs del backend o métricas de base de datos). |

### Pruebas funcionales

Las pruebas funcionales tienen como objetivo verificar que los servicios aprovisionados mediante OpenTofu se comporten de forma esperada al ejecutar las funcionalidades previstas. Se validará la capacidad del entorno de arquitectura de servicios para orquestar y ejecutar correctamente peticiones a través de sus distintos componentes: API Gateway, servicios backend, base de datos y almacenamiento local.

Estas pruebas simulan flujos reales del sistema desplegado (por ejemplo: registro, consulta, actualización y eliminación de entidades) con entrada de datos válidos e inválidos, validando la respuesta de los servicios expuestos y su integración.

Para ilustrar el comportamiento funcional, se utilizará la interfaz de Recooba como medio visual, conectada al backend desplegado, permitiendo ejecutar interacciones reales contra los servicios de arquitectura y verificar su funcionamiento extremo a extremo (end-to-end).

|  |  |
| --- | --- |
| **Elemento** | **Detalle** |
| **Objetivo de la prueba:** | Verificar que los servicios expuestos por la arquitectura respondan correctamente a las operaciones funcionales (registro, consulta, edición, eliminación), asegurando su coherencia e integridad bajo distintos escenarios. |
| **Técnica a utilizar:** | - Ejecución de casos de uso desde una interfaz cliente (ej. Recooba).- Validación de respuestas HTTP (códigos 200, 400, 500, etc.)- Envío de datos válidos e inválidos.- Simulación de errores y flujos interrumpidos. |
| **Criterio de validación:** | - Cada operación debe retornar el resultado esperado (OK, error, validación fallida, etc.)- El backend debe ejecutar correctamente la lógica de negocio sin comprometer integridad ni disponibilidad. |
| **Consideraciones especiales:** | - Las pruebas se realizarán con datos controlados.- Se recomienda usar Postman o cliente React Native conectado al backend desplegado.- Se deben revisar logs generados por los servicios para trazabilidad. |
| **Observaciones:** | Estas pruebas permiten comprobar la correcta operación del sistema distribuido y validar la coherencia del despliegue desde el cliente hasta la base de datos, como evidencia de que la arquitectura se comporta funcionalmente. |

### Pruebas de rendimiento (Performance)

|  |  |
| --- | --- |
| **Elemento** | **Detalle** |
| **Objetivo de la prueba:** | Evaluar el rendimiento y tiempo de respuesta de los servicios backend desplegados, midiendo su estabilidad bajo carga concurrente y su capacidad para mantener una operación fluida y predecible ante uso moderado. |
| **Técnica a utilizar:** | - Uso de herramientas como Apache Benchmark, Postman Collection Runner, o Artillery.io para simular múltiples peticiones.- Emulación desde la app Recooba conectada a backend real.- Registro de métricas. |
| **Criterio de validación:** | - Operaciones estándar deben tener tiempos de respuesta < **2 segundos**.- Operaciones pesadas (consultas masivas, escritura con imagen) no deben exceder **5 segundos**.- No debe haber caídas o errores 500. |
| **Consideraciones especiales:** | - Se deben ejecutar pruebas desde distintos dispositivos o entornos simulados (emuladores y físicos).- Es recomendable reiniciar los servicios entre ciclos para detectar problemas de acumulación o fuga de recursos. |
| **Observaciones:** | Estas pruebas ayudan a identificar cuellos de botella en la arquitectura y a validar que el sistema es escalable y responde adecuadamente cuando se incrementa el volumen de operaciones. |

### Pruebas de seguridad y de acceso a datos

|  |  |
| --- | --- |
| **Elemento** | **Detalle** |
| **Objetivo de la prueba:** | Verificar que los servicios desplegados permiten únicamente el acceso autorizado a los datos, que las credenciales se manejan de forma segura (con cifrado) y que se respetan los roles y permisos definidos en el backend. |
| **Técnica a utilizar:** | - Inyección de credenciales inválidas y sesiones caducadas.- Accesos simulados a rutas restringidas sin login.- Verificación de tokens JWT, cifrado de contraseñas y headers HTTPS.- Análisis con Postman, curl y logs del backend. |
| **Criterio de validación:** | - Solo usuarios autenticados pueden acceder a recursos protegidos.- Se deniega correctamente el acceso con tokens inválidos o ausentes.- Las contraseñas deben estar cifradas (bcrypt o equivalente). |
| **Consideraciones especiales:** | - El entorno debe ejecutar con HTTPS para validar certificados y transmisión segura.- Las pruebas deben cubrir intentos de acceso con cuentas inactivas o inexistentes.- Validar revocación o expiración de sesiones. |
| **Observaciones:** | Estas pruebas permiten comprobar la seguridad del backend desplegado, garantizando la confidencialidad y control de acceso en el contexto de una arquitectura escalable y segura para futuros entornos productivos. |

### Herramientas involucradas

|  |  |
| --- | --- |
| **Herramienta** | **Propósito** |
| **OpenTofu** | Aprovisionamiento de infraestructura como código (IaC) para entornos de prueba, test y desarrollo. |
| **Postman** | Ejecución de pruebas funcionales y de API sobre los endpoints expuestos por los servicios backend. |
| **Apache Benchmark (ab)** / **Artillery.io** | Simulación de carga para pruebas de rendimiento básicas sobre endpoints REST. |
| **DBeaver** / **MySQL Workbench** | Conexión y análisis de datos en la base MariaDB, para validar integridad y consultas SQL. |
| **Terminal (Bash / PowerShell)** | Ejecución de scripts, validación de despliegues y comandos de interacción directa con servicios. |
| **Git + GitHub / GitLab** | Control de versiones de los archivos de infraestructura, pruebas y documentación del proyecto. |
| **Visual Studio Code** | Edición de código backend, definición de pruebas y revisión de configuración de despliegue. |
| **Log system del backend** | Revisión de trazas para comprobar el flujo correcto de las peticiones y detección de errores. |
| **Recooba (cliente visual)** | Aplicación cliente usada como interfaz para validar los flujos funcionales contra la arquitectura desplegada. |

# Recursos

## Profesionales

|  |  |  |
| --- | --- | --- |
| **Rol** | **Responsabilidad** | **Especialización requerida** |
| **Arquitecto de Software** | Definir la estructura lógica y técnica de la arquitectura, establecer buenas prácticas de escalabilidad y modularidad, y supervisar la calidad de diseño. | Infraestructura como código (IaC), DevOps, OpenTofu, diseño técnico |
| **Ingeniero DevOps** | Encargado del despliegue automático del entorno mediante OpenTofu, integración continua y monitoreo del comportamiento de los servicios. | OpenTofu, CI/CD, bash scripting, administración de servidores |
| **QA / Tester de Arquitectura** | Diseñar y ejecutar pruebas funcionales, de rendimiento y seguridad sobre la arquitectura desplegada. Documentar resultados y hallazgos. | Postman, pruebas de API, análisis de rendimiento y seguridad |
| **Backend Developer** | Responsable de implementar y mantener los servicios consumidos por los clientes, así como gestionar su correcta exposición y modularidad. | Node.js, Express, REST, JWT, conexión a MariaDB |
| **Especialista en Base de Datos** | Validar la estructura, consistencia y seguridad del almacenamiento persistente. Realiza pruebas directas sobre MariaDB. | Modelamiento de datos, SQL, herramientas como DBeaver |
| **Observabilidad / Logging Analyst** | Monitorear el comportamiento del sistema mediante revisión de logs, alertas y dashboards. Identifica errores o cuellos de botella. | ELK Stack, Prometheus/Grafana (opcional), análisis de logs |

## Ambiente de pruebas

|  |  |  |
| --- | --- | --- |
| **Tipo de ambiente** | **Propósito** | **Características principales** |
| **Ambiente de Desarrollo (DEV)** | Espacio para ajustes iniciales en la definición de infraestructura, pruebas preliminares y debugging básico. | Aprovisionamiento completo con OpenTofu, acceso a consola y logs. Puede presentar cambios frecuentes. |
| **Ambiente de Pruebas Funcionales (QA)** | Validación estructurada de operaciones CRUD, autenticación, errores, control de acceso y comportamiento esperado. | Reproduce entorno de arquitectura real. Se ejecutan pruebas funcionales desde clientes como Postman o la app Recooba. |
| **Ambiente de Pruebas de Rendimiento (PERF)** | Ejecutar simulaciones de carga para medir respuesta del sistema ante múltiples usuarios o peticiones concurrentes. | Escenario aislado con herramientas de carga. Incluye métricas sobre CPU, memoria, tiempo de respuesta y estabilidad. |
| **Ambiente de Seguridad (SEC)** | Ejecutar pruebas sobre control de accesos, validación de sesiones, exposición de endpoints sensibles y cifrado. | Cubre aspectos como uso de HTTPS, tokens JWT, validación de roles y protección contra accesos no autorizados. |
| **Ambiente de Producción Simulado (UAT-like)** | Entorno final de pruebas integradas que simula condiciones reales antes de la puesta en marcha oficial. | Representa una réplica del entorno final. Útil para validación global antes del cierre de pruebas. |

### Preparación del ambiente de pruebas

La preparación del ambiente de pruebas se basa en la implementación controlada y automatizada de los distintos entornos requeridos, utilizando herramientas de infraestructura como código (IaC), en este caso, **OpenTofu**. Este enfoque permite levantar entornos desde cero, replicables y trazables, minimizando errores humanos y garantizando la homogeneidad entre pruebas.

El proceso de preparación considera las siguientes etapas:

1. **Clonación del repositorio IaC**: Se descarga el repositorio de configuración de infraestructura donde se encuentran los archivos .tf con la definición de recursos (servidores, red, base de datos, servicios, etc.).
2. **Ejecución de OpenTofu**: Se inicia el proceso de aprovisionamiento utilizando los comandos tofu init, tofu plan y tofu apply, lo que permite validar y desplegar todos los componentes definidos.
3. **Configuración de servicios backend**: Se levantan los servicios esenciales como la API REST (Node.js), la base de datos MariaDB, y herramientas de monitoreo o logs si se han definido.
4. **Verificación de endpoints**: Una vez desplegado, se valida la disponibilidad de los endpoints y recursos mediante pruebas iniciales (ping, curl, Postman) para asegurar su correcto funcionamiento.
5. **Integración con cliente de pruebas (Recooba)**: Si se requiere, se configura la app Recooba para que apunte a este entorno, permitiendo la ejecución de pruebas funcionales reales sobre la arquitectura.
6. **Población de datos de prueba**: Se insertan datos ficticios para simular distintos escenarios durante las validaciones (usuarios, registros, errores, etc.).
7. **Snapshot o backup del entorno base**: Una vez verificado, se guarda una copia del entorno para su reutilización o restauración si se detectan errores durante pruebas destructivas.

### Diseño del ambiente de pruebas

|  |  |  |
| --- | --- | --- |
| **ITEM** | **DESCRIPCIÓN** | **OBSERVACIONES** |
| **HARDWARE** |  |  |
| **Estación de Pruebas (Cliente)** |  |  |
| Procesador | Intel Core i5 2.5 GHz o superior |  |
| Memoria RAM | 8 GB DDR4 | Recomendado para ejecución fluida del emulador |
| Espacio en Disco | 100 GB SSD libre | Para simulador, logs, base de datos local |
| Tipo Monitor y Resolución | Full HD 1920x1080 | Compatible con diseño UI móvil |
| Unidad de Disquete | No aplica |  |
| Tarjeta de red | Gigabit Ethernet / Wi‑Fi 5 |  |
| Módem | Módem de red inalámbrica integrada |  |
| Mouse | Óptico estándar USB |  |
| Tipo Enlace | Conexión LAN o Wi‑Fi con mínimo 10 Mbps |  |
| **Servidores** |  |  |
| **Base de Datos (pruebas)** |  |  |
| Procesador | Intel Xeon o Ryzen 5 |  |
| Memoria RAM | 8 GB DDR4 |  |
| Espacio en Disco | 200 GB SSD |  |
| Tipo Monitor y Resolución | Acceso remoto, no requiere display físico |  |
| Unidad de Disquete | No aplica |  |
| Tarjeta de red | Gigabit Ethernet |  |
| Módem | No aplica |  |
| Mouse | No aplica |  |
| Tipo Enlace | VPN o LAN privada |  |
| **Aplicación (pruebas)** |  |  |
| Procesador | Igual a estación cliente |  |
| Memoria RAM | 8 GB |  |
| Espacio en Disco | 50 GB libres |  |
| Tipo Monitor y Resolución | Acceso desde emulador o dispositivo físico |  |
| Unidad de Disquete | No aplica |  |
| Tarjeta de red | Conectividad Wi‑Fi o datos móviles |  |
| **Impresoras** |  |  |
| Marca y Modelo | Epson L3250 o similar | Para impresión de reportes |
| Tipo | Multifuncional |  |
| Resolución | 5760 x 1440 dpi |  |
| Rendimiento | 33 ppm monocromo |  |
| Dedicación | Compartida |  |
| **RED** |  |  |
| Topología | Estrella |  |
| Medio | Inalámbrico y cableado |  |
| Velocidad | 100 Mbps o superior |  |
| Protocolo | TCP/IP |  |
| Módems | Integrado en router principal |  |
| Conexión Internet | Fibra óptica 400 Mbps |  |
| **Sistema de Respaldo / Restauración** |  |  |
| Unidad (Modelo y Marca) | Disco externo Seagate 1 TB |  |
| Capacidad | 1 TB SSD |  |
| Ubicación | Sala de servidores |  |
| **SOFTWARE** |  |  |
| **Estación de Pruebas (Cliente)** |  |  |
| Sistema Operativo | Windows 10 / Ubuntu 22.04 |  |
| Herramienta de testing | Jest, React Native Testing Library, Postman, Insomnia |  |
| Herramienta de Modelamiento | Draw.io, Lucidchart |  |
| BDMA | DBeaver + MariaDB 10.6 |  |
| Browser | Chrome, Edge, Firefox | Para acceso a portal web (si aplica) |
| Software de Escritorio | LibreOffice / Microsoft Office |  |
| **Base de Datos (pruebas)** |  |  |
| Sistema Operativo | Ubuntu Server 22.04 |  |
| Software de Red | MariaDB 10.6 configurado |  |
| Dominio/Cuenta | arquitectura\_test\_admin |  |
| **Aplicación (pruebas)** |  |  |
| Sistema Operativo | Linux (contenedores) / Android (si aplica visualización móvil) |  |
| Software de Red | HTTP/HTTPS vía API REST |  |
| Dominio/Cuenta | arquitectura\_app\_test |  |
| **Repositorios** |  |  |
| Servidor | GitHub | Repositorio principal |
| Dominio / Cuenta | github.com/renzogomez/arquitectura-garantias |  |
| **Seguridad** | VPN corporativa, contraseñas encriptadas, roles de acceso | Cumplimiento básico de seguridad de pruebas |

### Diseño ambiente de pruebas

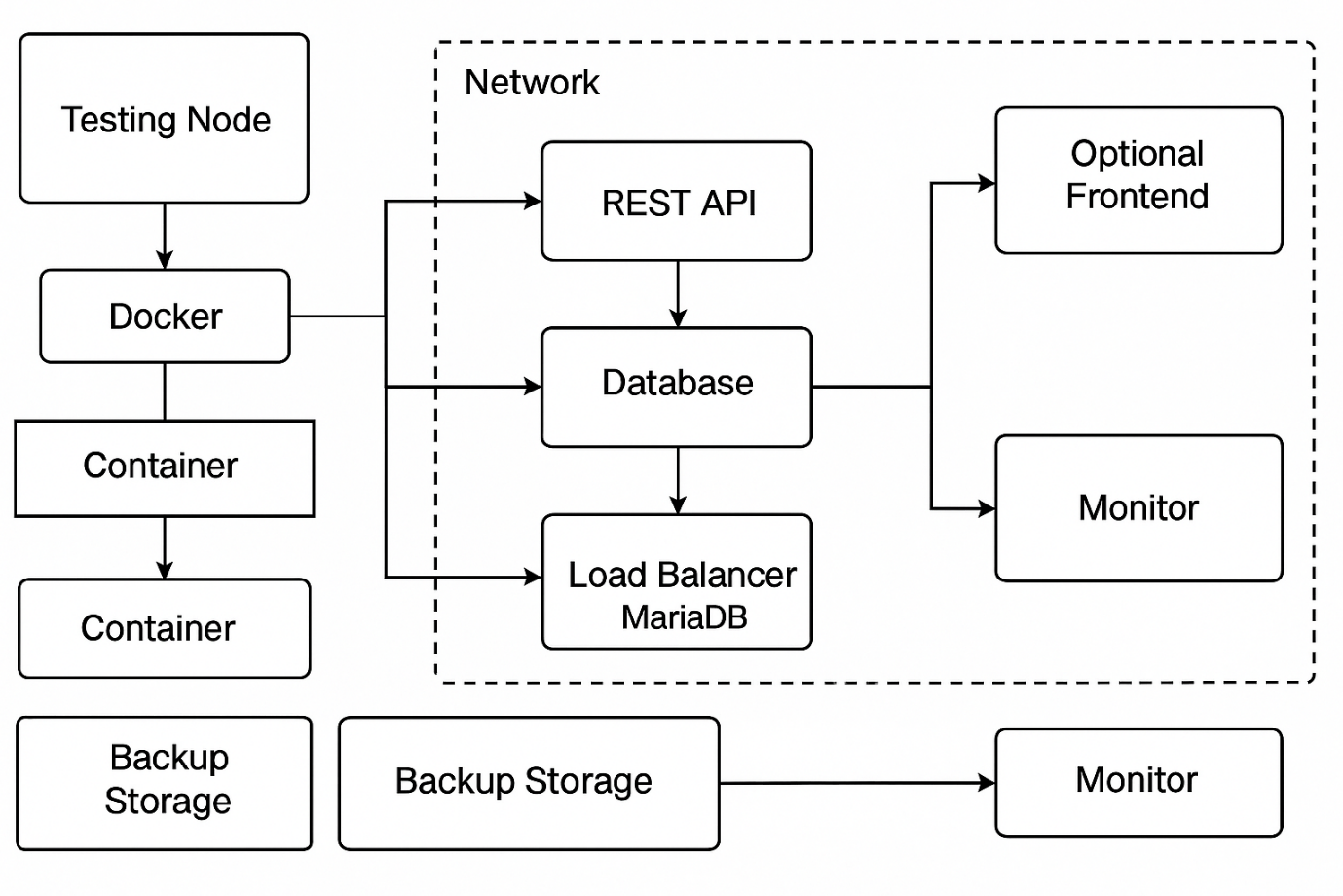
El presente diseño del ambiente de pruebas representa gráficamente la arquitectura técnica requerida para validar los distintos servicios que componen la solución escalable. Este entorno busca emular de forma controlada la infraestructura definida para producción, utilizando principios de *Infraestructura como Código* (IaC) y herramientas de aprovisionamiento como **OpenTofu** para desplegar y orquestar los componentes clave.

La arquitectura contempla una separación lógica de servicios por capas (aplicación, base de datos, red), integrando contenedores, balanceadores de carga y volúmenes de almacenamiento según sea necesario. El entorno está diseñado para ser reproducible, escalable y desacoplado, permitiendo ejecutar pruebas de estrés, rendimiento, recuperación ante fallos, despliegue automatizado y validación de configuración en escenarios reales de operación.

Este diseño considera:

* **Nodos de pruebas** desplegados con Docker o máquinas virtuales aprovisionadas desde código.
* **Servicios** como base de datos (MariaDB), APIs REST, frontends opcionales, gestores de logs, y herramientas de monitoreo.
* **Redes definidas como código** para asegurar aislamiento y conectividad controlada entre servicios.
* **Backups y recuperación** incluidos como parte del ciclo de prueba.

El diagrama asociado resume visualmente los elementos técnicos, sus conexiones y sus roles dentro del entorno de validación.



### Integración del ambiente de pruebas y configuración

La integración del ambiente de pruebas busca replicar, en la medida de lo posible, las condiciones de un entorno productivo para garantizar que los componentes desplegados bajo principios de Infraestructura como Código (IaC), mediante **OpenTofu**, funcionen de forma segura, estable y escalable.

Durante esta etapa, se coordina la correcta configuración de los distintos elementos que componen la arquitectura de servicios, como servidores backend, base de datos, almacenamiento persistente y conectividad segura, de forma que permitan validar la interacción de los módulos, la escalabilidad de los servicios y el comportamiento esperado ante distintos flujos de trabajo.

Las actividades generales a ser consideradas son:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Actividad** | **Responsable** | **Fecha Estimada** | **Fecha Real** | **Observaciones** |
| Aprovisionamiento de infraestructura con OpenTofu | Ingeniero DevOps | 01/06/2025 |  | Despliegue de servicios, redes, volúmenes y configuración declarativa (.tf) |
| Instalación de Base de Datos MariaDB (ambiente pruebas) | Administrador de BD | 02/06/2025 |  | Configuración de usuarios, backup, y verificación de integridad |
| Despliegue del backend de servicios (API REST) | Ingeniero Backend | 03/06/2025 |  | Validar conexión con BD, endpoints y variables de entorno |
| Implementación de control de acceso y seguridad | Ingeniero DevOps / QA | 04/06/2025 |  | Uso de roles, contraseñas encriptadas, y canal seguro (HTTPS + VPN) |
| Carga de datos simulados y verificación funcional | Tester Funcional | 05/06/2025 |  | Uso de scripts SQL o herramientas como DBeaver para datos ficticios |
| Prueba de conexión entre módulos y servicios | Tester QA / DevOps | 06/06/2025 |  | Validación de endpoints, logs, puertos y flujos entre componentes |
| Verificación de logs y monitoreo básico del entorno | Tester QA / Observabilidad | 06/06/2025 |  | Revisión de logs de servicio, respuesta ante errores, alertas básicas |

### Definición del banco de datos de prueba

Se deberán tomar las siguientes consideraciones al momento de generar el Banco de datos.

El banco de datos de prueba es una parte crítica del entorno de validación de la arquitectura, ya que permite simular escenarios reales de operación en una base de datos controlada y aislada. Esta base está estructurada en tablas equivalentes a las utilizadas por los servicios desplegados, pero poblada con información ficticia que permite evaluar la integridad, rendimiento, consultas y escritura de datos.

El aprovisionamiento se realiza en una instancia de **MariaDB** desplegada mediante **OpenTofu**, utilizando scripts de inicialización definidos en archivos .sql incluidos dentro del repositorio de infraestructura.

#### **Características del banco de datos de prueba:**

* **Base de datos:** test\_arch\_servicios
* **Motor:** MariaDB 10.6+
* **Tablas simuladas:** usuarios, servicios, logs, peticiones, respuestas, auditorias, entre otras.
* **Cantidad estimada de registros:** entre 500 y 2.000 entradas por tabla.
* **Relaciones y restricciones:** llaves primarias, foráneas y restricciones de integridad incluidas como en producción.
* **Contenido:**
  + Usuarios ficticios con distintos roles (admin, operador, auditor).
  + Logs de actividad con distintos niveles de severidad.
  + Entradas simuladas de peticiones API y sus respectivas respuestas.
  + Datos históricos para simular pruebas de recuperación, análisis y trazabilidad.

#### **Objetivos del banco de datos de prueba:**

* Validar las operaciones CRUD de cada servicio desplegado.
* Probar la respuesta del sistema frente a datos malformados o límites de ingreso.
* Comprobar la consistencia de los datos ante múltiples accesos simultáneos.
* Simular escenarios de uso intensivo o concurrencia para pruebas de rendimiento.
* Verificar la trazabilidad y logging de operaciones en condiciones controladas.

#### Profundidad

El diseño del banco de datos de prueba considera un nivel de profundidad suficiente para validar no solo el comportamiento individual de cada componente, sino también la interacción entre servicios dentro de una arquitectura modular y distribuida.

Se ha optado por una cobertura de datos que permita simular múltiples escenarios de uso, con distintos niveles de complejidad, tamaño y carga transaccional, incluyendo:

* **Casos normales:** Datos bien formados, relaciones correctas, y operaciones exitosas esperadas.
* **Casos límite:** Campos al máximo de su longitud, valores extremos (fechas, montos, tamaños de archivo), y uso simultáneo de múltiples servicios.
* **Casos erróneos o conflictivos:** Datos incompletos, claves inexistentes, referencias cruzadas inválidas, para probar validación, errores y manejo de excepciones.
* **Casos históricos:** Registros simulados que representan uso prolongado del sistema, crecimiento de logs, trazabilidad y auditoría de operaciones.

Esto asegura que el banco de pruebas pueda abordar no solo funcionalidades básicas, sino también aspectos críticos como **resiliencia, tolerancia a fallos, rendimiento, concurrencia** y capacidad de recuperación, todos esenciales en entornos orientados a microservicios y escalabilidad.

Además, la definición declarativa del entorno a través de **OpenTofu** permite generar versiones del banco de datos en diferentes estados (inicial, cargado, con errores, etc.), facilitando pruebas repetibles y consistentes en diferentes etapas del ciclo de validación.

#### Variación

El banco de datos de prueba contempla un conjunto variado de datos diseñados para evaluar el comportamiento de la arquitectura ante distintos tipos de entradas, estructuras y cargas de trabajo. Esta variación es fundamental para validar la robustez, adaptabilidad y estabilidad de los servicios desplegados, permitiendo simular condiciones diversas que podrían presentarse en entornos reales.

#### **Tipos de variación considerados:**

* **Variación estructural:**
  + Combinaciones de registros completos e incompletos.
  + Diferentes tamaños de datos (campos vacíos, medianos y al límite de longitud).
  + Tablas con relaciones 1:1, 1:N y N:M con distintos niveles de profundidad jerárquica.
* **Variación temporal:**
  + Registros con fechas pasadas, actuales y futuras.
  + Datos creados en distintos periodos simulados para evaluar comportamiento histórico.
* **Variación lógica:**
  + Usuarios con distintos roles y niveles de acceso (admin, operador, consulta).
  + Registros con comportamientos esperados y no esperados para probar validaciones.
* **Variación de carga:**
  + Tablas con bajo volumen (menos de 10 registros) y otras con cientos o miles.
  + Simulación de alta concurrencia mediante scripts de prueba automatizados.
* **Variación de escenarios negativos:**
  + Datos inconsistentes para forzar errores de integridad referencial.
  + Entradas con formatos incorrectos o valores fuera de rango.

Esta diversidad de datos permite que las pruebas no se limiten al funcionamiento ideal, sino que consideren casos de borde, malformaciones y escenarios caóticos. Así se garantiza que la arquitectura sea capaz de **responder adecuadamente bajo cualquier condición**, y que los mecanismos de validación, recuperación, logs y alertas funcionen correctamente.

#### Alcances

El banco de datos de prueba está diseñado para cubrir de forma amplia los distintos componentes y capas de la arquitectura desplegada, permitiendo validar tanto funcionalidades internas como interacciones entre servicios, seguridad y comportamiento ante distintas cargas.

#### **Alcance general:**

* **Cobertura de servicios:**
  + Todas las APIs expuestas por el backend.
  + Módulos internos como autenticación, auditoría, gestión de usuarios y logs.
  + Componentes de almacenamiento, procesamiento y respuesta.
* **Capas validadas:**
  + **Persistencia:** pruebas directas sobre la base de datos (integridad, relaciones, escritura, lectura).
  + **Lógica de negocio:** validación de reglas implementadas en los servicios.
  + **Comunicación:** validación de solicitudes/respuestas entre servicios vía API REST o eventos.
  + **Seguridad:** pruebas de acceso según roles y validación de tokens.
* **Escenarios contemplados:**
  + Alta concurrencia en servicios clave.
  + Errores controlados en endpoints y sus respuestas.
  + Validación de consistencia tras operaciones múltiples.
  + Simulación de caídas o errores para validar mecanismos de recuperación.

#### **Limitaciones del alcance:**

* No se contempla en esta fase el uso de datos sensibles o reales de usuarios finales.
* No se incluyen pruebas de recuperación de desastres a gran escala (respaldo y restauración total).
* Pruebas de integración con servicios externos (como cloud provider, servicios de terceros) serán cubiertas en un entorno posterior de staging o preproducción.

Este alcance permite asegurar que la arquitectura definida no solo es funcional, sino también **estable, flexible y segura**, cumpliendo los requisitos de escalabilidad y mantenibilidad definidos para el proyecto.

#### Ejecución

La ejecución del banco de datos de prueba se realizará de forma automatizada al inicio de cada ciclo de pruebas, asegurando condiciones consistentes, controladas y repetibles. Este proceso es fundamental para validar correctamente el comportamiento de la arquitectura bajo diferentes escenarios y garantizar la confiabilidad de los resultados.

#### **Mecanismo de ejecución:**

* Se emplearán **scripts declarativos en SQL y/o archivos .tf asociados a OpenTofu** para la creación, limpieza y carga de datos en la instancia de prueba.
* Los scripts son ejecutados automáticamente al levantar el entorno de prueba, como parte del pipeline de despliegue o mediante tareas programadas previas a cada suite de validación.
* Cada ejecución asegura que:
  + Se eliminan datos residuales de pruebas anteriores.
  + Se restablece el esquema base, incluyendo relaciones, restricciones y usuarios.
  + Se cargan los datos requeridos según el escenario de prueba (normal, límite, carga, error).

#### **Herramientas de apoyo:**

* Scripts .sh o .ps1 para automatizar la conexión y ejecución de comandos sobre la base de datos MariaDB.
* Contenedores Docker o módulos OpenTofu para levantar una instancia temporal de base de datos.
* Conexiones verificadas desde herramientas como **DBeaver, Insomnia, Postman o Jest** para lanzar validaciones de forma integrada.

#### **Frecuencia de ejecución:**

* Antes de cada ciclo formal de pruebas (funcionales, integración, carga, etc.).
* Al validar un nuevo módulo de la arquitectura.
* Después de aplicar cambios a los esquemas o servicios que dependan de datos estructurados.

Este enfoque asegura que **la ejecución del banco de pruebas sea confiable, reproducible y escalable**, eliminando dependencias manuales y minimizando errores humanos durante el proceso de verificación.

#### Condiciones

Para que el banco de datos de prueba cumpla su propósito de validar correctamente el comportamiento de la arquitectura de servicios, es necesario contar con documentación técnica detallada que permita definir registros representativos y consistentes.

#### **Documentación requerida:**

* **Definición de Requerimientos Arquitectónicos:**
  + Incluye tanto los requerimientos funcionales como los no funcionales (seguridad, escalabilidad, disponibilidad, etc.) que orientan el diseño de los datos de prueba.
* **Esquema de Base de Datos y Modelo Entidad-Relación (ER):**
  + Representación formal de las entidades, relaciones, claves primarias/foráneas, restricciones de integridad, y estructuras específicas que sustentan los servicios desplegados.
  + Este esquema debe reflejar la arquitectura modular y los componentes desacoplados que se comunican mediante APIs o eventos.
* **Diagramas de Casos de Uso a Nivel de Arquitectura:**
  + Permiten identificar las interacciones entre los distintos actores (usuarios, servicios, sistemas externos) y los puntos críticos donde los datos deben estar disponibles o ser validados.
* **Diagrama de Estados de Componentes Críticos:**
  + Especifican los distintos estados posibles de las entidades del sistema (por ejemplo, estados de una solicitud, transacción o sesión) y ayudan a construir escenarios de prueba coherentes.

Estas condiciones son necesarias para garantizar que los datos generados en el banco de pruebas representen con fidelidad los posibles flujos, errores, excepciones y estados del sistema. Además, permiten mantener la **alineación entre los datos de prueba y los objetivos de calidad arquitectónica**, tales como resiliencia, trazabilidad, interoperabilidad y consistencia.

### Generación de datos

La generación de datos para las pruebas dentro de esta arquitectura de servicios se diseña con el objetivo de garantizar la cobertura adecuada de escenarios funcionales y no funcionales, incluyendo casos normales, casos límite y situaciones excepcionales. La calidad de estos datos es esencial para obtener resultados válidos que permitan evaluar la robustez, rendimiento, integridad y resiliencia del sistema.

#### **Aspectos considerados en la generación de datos:**

* **Validez estructural y semántica:**
  + Los datos deben respetar el esquema definido en el modelo entidad-relación, cumpliendo con claves primarias, foráneas, restricciones de unicidad y formatos de campo.
  + Deben representar escenarios reales de operación, manteniendo coherencia entre módulos.
* **Cobertura de casos de uso y estados del sistema:**
  + Se incluirán datos que simulen entradas válidas e inválidas, errores de usuario, flujos alternativos, estados intermedios de procesamiento y condiciones de borde.
* **Automatización de la carga:**
  + Se utilizarán scripts automatizados (bash, SQL, Terraform/OpenTofu, etc.) para poblar la base de datos antes de cada ejecución de pruebas, garantizando un entorno limpio y controlado.
  + Los datos estarán almacenados en archivos estructurados (.sql, .json, .csv), versionados junto al código fuente en el repositorio.
* **Diversificación de condiciones:**
  + Incluir datos con distintos perfiles de usuario (administrador, operador, invitado).
  + Registros que incluyan fechas pasadas, actuales y futuras.
  + Variación en la cantidad de registros (desde pocos hasta miles) para evaluar rendimiento.
* **Seguridad y anonimización:**
  + No se utilizarán datos reales de usuarios.
  + Los registros estarán compuestos por datos ficticios diseñados exclusivamente para pruebas.

La generación de datos busca, en definitiva, que cada prueba realizada esté soportada por un conjunto representativo y confiable de información, lo que permite evaluar el desempeño de la arquitectura tanto en condiciones normales como en situaciones críticas o inesperadas.

#### Muestra de producción

Para validar la arquitectura bajo condiciones más cercanas a la operación real, se incorporará una **muestra controlada de datos similares a los que se utilizarían en producción**, sin comprometer la seguridad ni la privacidad de usuarios reales.

#### **Objetivo:**

Evaluar el comportamiento de los componentes de la arquitectura en un entorno que simule condiciones reales de volumen, complejidad y distribución de datos, lo que permitirá detectar cuellos de botella, errores lógicos y posibles brechas en el rendimiento o en la comunicación entre servicios.

#### **Características de la muestra:**

* **Datos simulados, no reales:**
  + Aunque emulan fielmente estructuras y flujos productivos, los datos utilizados son generados artificialmente.
  + Se crean registros que reproducen los patrones, proporciones y relaciones esperadas en producción.
* **Escenarios de uso representativos:**
  + Se incluyen casos frecuentes, inusuales y extremos.
  + Se simulan condiciones de carga moderada a alta, con múltiples usuarios, solicitudes simultáneas y estados de transición entre servicios.
* **Distribución realista:**
  + Registros de diferentes fechas, entidades geográficas, tipos de usuario, y estados del sistema.
  + Estructura jerárquica de datos para simular relaciones entre módulos o dominios (por ejemplo, usuarios → operaciones → auditorías).
* **Persistencia y trazabilidad:**
  + Se monitorea el comportamiento de la arquitectura ante operaciones sobre estos datos: consultas, inserciones, actualizaciones y eliminaciones.
  + Se verifica que la trazabilidad (logs, auditorías) funcione correctamente.
* **Aislada del entorno productivo:**
  + Toda la ejecución ocurre en un entorno de pruebas independiente, sin interacción con bases de datos reales o servicios en producción.
  + Los datos no son migrados ni replicados desde ambientes reales.

Este enfoque permite comprobar si la arquitectura definida es **capaz de soportar escenarios reales de negocio**, asegurando que los componentes estén preparados para una futura implementación en un entorno productivo más exigente.

#### Datos complementarios

Durante el proceso de pruebas, se contempla la incorporación de **datos adicionales** que no forman parte del flujo central de la arquitectura, pero que son necesarios para validar condiciones específicas, simular entornos externos o ejecutar escenarios que requieren condiciones especiales.

#### **Tipos de datos complementarios considerados:**

* **Datos ficticios de terceros o sistemas externos:**
  + Se generan registros simulados que representan respuestas esperadas de servicios con los que la arquitectura podría integrarse en el futuro (como sistemas de pago, servicios de identidad, almacenamiento externo, etc.).
* **Datos de configuración y metainformación:**
  + Incluye parámetros de entorno, configuraciones de red, archivos .env, claves de acceso, tokens simulados, certificados y variables utilizadas por la infraestructura como código (IaC).
* **Datos para pruebas de carga y estrés:**
  + Se preparan registros adicionales en volumen masivo para someter la arquitectura a condiciones de uso intensivo.
  + Estos datos permiten validar la respuesta del sistema ante operaciones concurrentes o saturación de servicios.
* **Datos aleatorios o no esperados:**
  + Se incluyen entradas no válidas, incompletas o inconsistentes para evaluar los mecanismos de validación, logs y respuestas de error del sistema.
* **Datos históricos simulados:**
  + Registros con fechas antiguas o con referencias a flujos ya cerrados, que permiten probar funciones de auditoría, reportes y archivado de información.

Los datos complementarios cumplen una función crítica: **garantizar que el sistema sea robusto, tolerante a errores, interoperable y seguro** incluso en situaciones no contempladas por los flujos estándar. Su uso estratégico permite elevar el nivel de cobertura y confianza en la validación de la arquitectura.

#### Envejecimiento de datos de prueba

El **envejecimiento de datos de prueba** es un proceso clave para simular condiciones reales en las que los datos han permanecido en el sistema durante un período prolongado, permitiendo validar el comportamiento de la arquitectura frente a:

* **Procesamiento de grandes volúmenes históricos.**
* **Manejo de registros obsoletos o inactivos.**
* **Estrategias de archivado, limpieza o compresión de datos.**

#### **Objetivos del envejecimiento de datos:**

* Evaluar el rendimiento de consultas sobre tablas con millones de registros.
* Validar la integridad de datos antiguos mantenidos por servicios de largo ciclo de vida.
* Comprobar si los mecanismos de notificación, auditoría o mantenimiento responden correctamente al paso del tiempo.
* Asegurar que no existan fugas de memoria, cuellos de botella o degradaciones en la arquitectura ante el uso continuo.

#### **Técnicas aplicadas:**

* **Manipulación de fechas en scripts de carga:**
  + Se generan registros con fechas de creación, modificación o vencimiento de hasta 2 o más años en el pasado.
* **Simulación de ciclos completos de uso:**
  + Inserción de datos que ya pasaron por todo el flujo del sistema (creación → uso → caducidad → archivo).
* **Tareas programadas y eventos automáticos:**
  + Se activan funciones que dependen del paso del tiempo, como limpieza de logs, envío de alertas por expiración, o inactivación automática de servicios.

#### **Validaciones esperadas:**

* El sistema debe mantener estabilidad y tiempos de respuesta aceptables al consultar datos envejecidos.
* Las reglas de negocio y restricciones temporales deben respetarse con exactitud.
* Se debe garantizar la trazabilidad completa, incluso sobre datos antiguos.

Este enfoque permite simular el comportamiento del sistema en contextos más cercanos al uso real prolongado, aportando evidencias clave sobre su **escalabilidad, mantenibilidad y resistencia a la degradación con el tiempo**.

#### Procesos batch

Se consideran algunos procesos batch necesarios para la actualización de datos:

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **Job** | **Archivo Maestro** | **Archivo de Movimiento** | **Tablas** | **Interfaces** | **Informes** |
| job\_backup\_diario.sh | backup\_config.yaml | backup\_estado.log | logs, usuarios, servicios | API almacenamiento remoto (S3) | Log de respaldo diario, informe de estado en PDF |
| job\_purge\_logs.sh | reglas\_purga.json | purge\_resultado.log | logs, auditoria | None | Registro de purga, resumen de espacio liberado |
| job\_alertas\_expiracion.py | alertas\_config.json | alertas\_ejecucion.log | garantias, usuarios | Notificaciones internas (SMTP, API SMS) | Listado de alertas enviadas, errores y pendientes |
| job\_migracion\_datos.sh | migracion\_schema.yaml | migracion\_log.csv | clientes, garantias\_hist | Conexión entre entornos (staging/prod) | Informe de datos migrados, resumen de cambios |
| job\_cierre\_mensual.sql | cierre\_config.sql | cierre\_resultado.log | movimientos, resumen\_cierre | API contable | PDF con KPIs mensuales, errores de cierre detectados |

# Actividades e Hitos del Plan de Pruebas

Plan de Pruebas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tarea** | **Responsable** | **Esfuerzo (días)** | **Fecha Inicio** | **Fecha Término** |
| Identificar el Proyecto | Jefe de Proyecto | 1 | 06-may-2025 | 06-may-2025 |
| Definir Estrategia | Líder QA | 2 | 07-may-2025 | 08-may-2025 |
| Estimar Actividades | Líder QA | 1 | 09-may-2025 | 09-may-2025 |
| Identificar Recursos | Líder QA | 1 | 10-may-2025 | 10-may-2025 |
| Documentar el “Plan de Pruebas” | Líder QA | 2 | 13-may-2025 | 14-may-2025 |
| Agendar de Actividades | Jefe de Proyecto | 1 | 15-may-2025 | 15-may-2025 |
| Revisar el “Plan de Pruebas” | Stakeholders | 2 | 16-may-2025 | 17-may-2025 |

Diseño de las Pruebas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tarea** | **Responsable** | **Esfuerzo (días)** | **Fecha Inicio** | **Fecha Término** |
| Analizar Requerimientos | Analista QA | 2 | 20-may-2025 | 21-may-2025 |
| Especificar Procedimientos de Prueba | Analista QA | 2 | 22-may-2025 | 23-may-2025 |
| Especificar Casos de Prueba | Analista QA | 2 | 24-may-2025 | 27-may-2025 |
| Revisar Cobertura de Requisitos | Líder QA | 1 | 28-may-2025 | 28-may-2025 |

Implementación de las Pruebas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tarea** | **Responsable** | **Esfuerzo (días)** | **Fecha Inicio** | **Fecha Término** |
| Establecer Ambiente de Implementación | Admin. de Sistemas | 2 | 29-may-2025 | 30-may-2025 |
| Desarrollar Procedimientos de Prueba | Desarrollador/Implementador | 3 | 31-may-2025 | 04-jun-2025 |
| Probar y Depurar los Procedimientos | QA + Desarrollador | 3 | 05-jun-2025 | 07-jun-2025 |
| Modificar los Procedimientos | Desarrollador | 2 | 10-jun-2025 | 11-jun-2025 |

Ejecución de las Pruebas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tarea** | **Responsable** | **Esfuerzo (días)** | **Fecha Inicio** | **Fecha Término** |
| Ejecutar Pruebas | Tester | 5 | 12-jun-2025 | 18-jun-2025 |
| Comprobar Resultados Esperados | Tester | 2 | 19-jun-2025 | 20-jun-2025 |
| Investigar Resultados Inesperados | Tester + Desarrollador | 3 | 21-jun-2025 | 24-jun-2025 |
| Registrar Defectos (Log) | Tester | 1 | 25-jun-2025 | 25-jun-2025 |
| Re-Ejecutar las Pruebas | Tester | 2 | 26-jun-2025 | 27-jun-2025 |

Evaluación de las Pruebas

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tarea** | **Responsable** | **Esfuerzo (días)** | **Fecha Inicio** | **Fecha Término** |
| Revisar el Log de Pruebas | Líder QA | 1 | 01-jul-2025 | 01-jul-2025 |
| Evaluar Cobertura de los Casos | Líder QA | 1 | 02-jul-2025 | 02-jul-2025 |
| Evaluar Defectos | Líder QA | 1 | 03-jul-2025 | 03-jul-2025 |
| Reportar Defectos | Líder QA | 2 | 04-jul-2025 | 07-jul-2025 |

Cierre y Consolidación

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Tarea** | **Responsable** | **Esfuerzo (días)** | **Fecha Inicio** | **Fecha Término** |
| Consolidar Resultados Finales | Líder QA | 2 | 08-jul-2025 | 09-jul-2025 |
| Presentación del Informe Final | Jefe de Proyecto | 2 | 10-jul-2025 | 11-jul-2025 |
| Aprobación de Resultados | Stakeholders | 1 | 12-jul-2025 | 12-jul-2025 |
| Cierre del Plan de Pruebas | Líder QA | 1 | 14-jul-2025 | 14-jul-2025 |

# Entregables

En esta sección se listan los documentos, herramientas y reportes que serán creados como parte del proceso de pruebas, junto con sus criterios de entrada, salida, suspensión y los mecanismos para reportar resultados y defectos.

## Plan de pruebas

Los siguientes entregables forman parte del Plan de Pruebas de la **arquitectura de servicios orientada a la gestión digital de garantías mediante notificaciones calendarizadas**, sobre la cual se sustenta el caso de uso funcional Recooba:

* Documento “Plan de Pruebas”
* Casos de prueba documentados en la “Planilla de Casos de Prueba”
* Resumen de resultados en el “Sumario de Evaluación de Pruebas”
* Reportes de defectos

### Criterio de entrada para el “Plan de pruebas”

La ejecución del plan de pruebas podrá comenzar cuando se cumplan las siguientes condiciones:

* El entorno de pruebas ha sido configurado y validado correctamente.
* La arquitectura ha sido desplegada con una versión funcional en el entorno de pruebas (incluyendo infraestructura, base de datos, APIs y servicios).
* Se dispone de los casos de prueba documentados y revisados.
* Los datos de prueba han sido generados y cargados según las especificaciones.
* El equipo de QA ha sido asignado y está disponible.

### Criterio de salida para el “Plan de pruebas”

Se considerará que el plan de pruebas ha finalizado exitosamente cuando:

* Se hayan ejecutado el 100% de los casos de prueba definidos para los componentes de la arquitectura.
* Todos los defectos críticos y de alta prioridad hayan sido resueltos y verificados.
* Se haya emitido el informe de resultados finales de pruebas.
* Se haya realizado una reunión de cierre de pruebas con aprobación del equipo de QA y los stakeholders técnicos del proyecto.

### Criterio de suspensión y resumisión

La ejecución del plan de pruebas podrá suspenderse en los siguientes casos:

* Se detectan errores críticos que impiden continuar las pruebas (en servicios, integraciones, base de datos, etc.).
* Fallas de infraestructura o disponibilidad de entornos.
* Falta de recursos clave del equipo QA o herramientas.

Las pruebas se podrán reanudar cuando:

* Los errores críticos hayan sido corregidos y validados.
* El entorno haya sido restablecido.
* Se asegure la disponibilidad del equipo y herramientas necesarias.

## Resultados de las pruebas

Los resultados de las pruebas se registrarán utilizando las siguientes herramientas:

* **Planilla de Casos de Prueba**: para registrar el estado de cada caso (Pasó, Falló, En ejecución) asociado a los distintos componentes y servicios de la arquitectura.
* **JIRA**: para documentar y dar seguimiento a los defectos detectados en APIs, servicios, base de datos u otros módulos técnicos.
* **Informe Final de Pruebas**: documento que consolida el cumplimiento de los casos de prueba, los defectos detectados y las conclusiones de la validación arquitectónica.
* **Bitácora de pruebas**: registro diario de ejecución de pruebas y hallazgos técnicos.

Los resultados se evaluarán en función de:

* Porcentaje de componentes de la arquitectura validados.
* Porcentaje de pruebas exitosas (funcionales, de integración, de rendimiento, etc.).
* Cantidad y severidad de defectos encontrados en los diferentes niveles de la arquitectura.

## Reporte de defectos

El reporte de defectos es un componente clave del proceso de aseguramiento de calidad dentro del plan de pruebas de la arquitectura de servicios. Su objetivo es registrar, clasificar, hacer seguimiento y resolver los errores detectados durante la ejecución de los distintos tipos de pruebas (unitarias, de integración, funcionales, de rendimiento, entre otras).

Cada defecto identificado debe incluir la siguiente información mínima:

* **ID del Defecto**: Código único para su trazabilidad.
* **Descripción**: Detalle claro y conciso del comportamiento observado.
* **Componente Afectado**: Módulo o servicio específico dentro de la arquitectura.
* **Severidad**: Crítica, Alta, Media o Baja, según impacto en el sistema.
* **Prioridad**: Urgente, Alta, Media o Baja, según necesidad de corrección.
* **Ambiente**: Entorno donde fue detectado (QA, staging, integración).
* **Evidencia**: Capturas, logs o mensajes de error relevantes.
* **Estado**: Reportado, En análisis, Asignado, Resuelto, Cerrado.
* **Responsable**: Rol o integrante encargado de resolver el defecto.

Los defectos serán gestionados a través de herramientas como **JIRA**, lo que permite su seguimiento por parte del equipo técnico, priorización conjunta con los stakeholders y registro de métricas de calidad tales como:

* Porcentaje de defectos críticos resueltos.
* Tiempo promedio de resolución.
* Distribución de defectos por componente o servicio.
* Tendencias por iteración o sprint.

Además, los defectos cerrados serán validados mediante pruebas de regresión específicas, asegurando que las correcciones no afecten negativamente a otros componentes de la arquitectura.

# Anexos

## A: Tareas del proyecto

La siguiente lista muestra las tareas relacionadas con el “Plan de pruebas”:

|  |  |
| --- | --- |
| *✓* | “Plan de pruebas” (preliminar, al inicio del proyecto) |
| *✓* | Identificar requerimientos para el testing |
| *✓* | Identificar los riesgos, cuantificar impacto |
| *✓* | Desarrollar la estrategia de pruebas |
| *✓* | Identificar los recursos para las pruebas |
| *✓* | Generar “Plan de Pruebas” detallado |
| *✓* | Diseño general de las pruebas |
| *✓* | Análisis de carga |
| *✓* | Identificar y describir los casos de prueba |
| *✓* | Identificar y estructurar los procedimientos de prueba |
| *✓* | Revisar y accesar la cobertura de las pruebas |
| *✓* | Implementar las pruebas |
|  | Grabar o programar los *scripts* de las pruebas, si aplica |
|  | Identificar las funcionalidades a probar, específicos en el modelo de diseño e implementación |
|  | Establecer el conjunto de datos externos |
| *✓* | Ejecutar las pruebas |
| *✓* | Ejecutar los procedimientos de prueba |
| *✓* | Evaluar la ejecución de las pruebas |
| *✓* | Comprobar los resultados |
| *✓* | Investigar los resultados inesperados |
| *✓* | Registro de defectos, Informe de Resultados |
| *✓* | Evaluar las pruebas |
| *✓* | Evaluar la cobertura de los casos de prueba |
| *✓* | Evaluar la cobertura del código |
| *✓* | Analizar defectos |

## B: Pruebas de rendimiento (*performance*)

*El testing de rendimiento es una prueba para conocer los tiempos de respuesta, la tasa de transacciones y otros tiempos requeridos de desempeño. Su objetivo es comprobar que los tiempos de respuesta requeridos son los óptimos, y que el desempeño del sistema y la configuración del hardware son adecuados frente a una serie de transacciones bajo ciertas condiciones de carga.*

Para esto, se ejecutarán transacciones específicas que simulan el uso del sistema por parte de un usuario final, bajo dos condiciones principales:

* Carga normal de trabajo
* Sobrecarga o estrés del sistema

|  |  |
| --- | --- |
| Objetivo de la prueba: | Verificar el comportamiento de los servicios desplegados bajo diferentes niveles de carga, validando el tiempo de respuesta, estabilidad, consumo de recursos y cumplimiento de los SLA definidos. Las condiciones a evaluar incluyen:   * Carga normal de trabajo * Sobrecarga o estrés del sistema |
| Técnica a usar: | * **Reutilización de scripts de pruebas automatizadas**, ajustados para simular peticiones concurrentes a los endpoints expuestos. * **Generación de carga artificial** con herramientas como Apache JMeter, k6, Artillery o locust.io. * **Ejecución de pruebas en entornos escalables**, replicando nodos o contenedores para simular múltiples clientes simultáneos. * **Monitoreo en tiempo real** con Prometheus, Grafana o CloudWatch (según el entorno). |
| Criterio de validación: | * Cada transacción debe completarse dentro del tiempo máximo aceptado (SLA). * **El sistema no debe fallar ni degradar servicios críticos**, incluso con múltiples usuarios simultáneos. * Se deben detectar **cuellos de botella o puntos de saturación**. |
| Consideraciones especiales: | * Las pruebas deben ejecutarse en un **ambiente controlado y replicable**. * La base de datos y APIs deben contener datos equivalentes en tamaño y distribución a los de producción.   Las pruebas deben cubrir:   * Escenario de carga estable * Escenario de estrés gradual * Escenario de ruptura controlada |
| Observaciones: | * Se recomienda **documentar cada ejecución**, registrando tiempos de respuesta, uso de CPU/RAM, IOPS y latencia. * Ideal ejecutar pruebas en **horarios fuera de ventana operativa**, si se comparte infraestructura. * Resultados deben ser analizados con herramientas de monitoreo y logs centralizados. |

## C: Pruebas de seguridad y de control de acceso

Se recomienda que el Administrador de la Red y del Sistema planifique y participe en la ejecución de este tipo de pruebas.

Este *testing* se enfoca en dos áreas claves:

* **Seguridad a nivel de la Aplicación:** Verifica que los usuarios sólo accedan a funciones o datos permitidos según su perfil o rol.
* **Seguridad a nivel del Sistema:** Evalúa el acceso general al sistema, incluyendo autenticación (*login*) y conexión remota.

|  |  |
| --- | --- |
| Objetivo de la prueba: | * Validar que **solo usuarios autorizados** puedan acceder a los servicios expuestos por la arquitectura. * Comprobar que se respetan los niveles de acceso por **rol, token o credencial** definidos en la política de autenticación/autorización. * Verificar que los sistemas de backend y API Gateway bloqueen correctamente accesos indebidos. |
| Técnica a usar: | * **Ejecución de pruebas de acceso** simulando distintos perfiles de usuario (admin, usuario final, anónimo). * **Intentos de acceso forzado o fuera de rol** a través de herramientas como Postman, curl, Burp Suite o OWASP ZAP. * Pruebas con **tokens JWT expirados o manipulados**, sesiones inválidas y credenciales incorrectas. * Verificación del correcto funcionamiento del sistema ante múltiples intentos fallidos (brute force, token replay, etc). |
| Criterio de validación: | * El sistema debe **denegar todo intento no autorizado** con códigos HTTP adecuados (401, 403). * Solo los usuarios con los privilegios correspondientes deben acceder a sus recursos. * Los eventos de autenticación, denegación y acceso deben **quedar registrados en logs de auditoría**. |
| Consideraciones especiales: | * Las pruebas deben realizarse sobre un entorno **que replique la configuración de seguridad**: políticas de IAM, reglas del API Gateway, roles RBAC, etc. * Requiere la participación de un administrador de sistemas o DevOps para observar logs en tiempo real. * Es ideal utilizar un entorno con **sistemas de monitoreo de amenazas y alertas activas** (por ejemplo, WAF, CloudTrail, Falco, etc.). |
| Observaciones: | * **Todos los accesos deben ser registrados**, incluyendo logs de intentos fallidos y exitosos. * Se deben evaluar tanto accesos vía **frontend** (si aplica) como accesos directos a **servicios REST, gRPC o CLI/API internas**. * Se recomienda integrar estas pruebas como parte del pipeline CI/CD para entornos de staging o preproducción. |