Desarrollo de una herramienta de pruebas para aplicaciones basadas en eventos que usan RabbitMQ

Ever Daniel Rivera Inagán

Proyecto de grado entregado para obtener el título de
Magister en Ingeniería de Software

Dirigida por
Msc. Juan Pablo García Cifuentes

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería y Ciencias
Maestría en Ingeniería de Software
Santiago de Cali
30 de Noviembre de 2025

Santiago de Cali, 30 de Noviembre de 2025.

Señores
Pontificia Universidad Javeriana Cali.
Ph.D. Luisa Rincón
Directora Maestría en Ingeniería de Software.
Cali.

Cordial Saludo.

Por medio de la presente hago constar que en mi calidad de director de trabajo de grado he revisado el proyecto titulado "Desarrollo de una herramienta de pruebas para aplicaciones basadas en eventos que usan RabbitMQ" realizado por el estudiante de Magister en Ingeniería de Software Ever Daniel Rivera Inagán (cod: 0221267), el cual se encuentra terminado y considero que cumple con los requisitos para ser sustentado.
Atentamente,

Juan Pablo García Cifuentes

Firmado digitalmente por Juan Pablo García Cifuentes
Fecha: 2026.02.17 21:55:17 -05'00'

Msc. Juan Pablo García Cifuentes

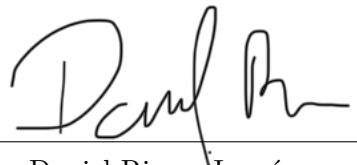
Santiago de Cali, 30 de Noviembre de 2025.

Señores
Pontificia Universidad Javeriana Cali
Ph.D. Luisa Rincón
Directora Maestría en Ingeniería de Software
Cali.

Cordial Saludo.

Me permito presentar a su consideración el proyecto de grado titulado "Desarrollo de una herramienta de pruebas para aplicaciones basadas en eventos que usan RabbitMQ" con el fin de cumplir con los requisitos exigidos por la Universidad y para que sea sometido a revisión del jurado y cumpla su aprobación, para conseguir posteriormente el título de Magister en Ingeniería de Software.

Atentamente,



Ever Daniel Rivera Inagán
Código: 0221267

Ficha Resumen
Trabajo de Grado Maestría en Ingeniería de Software

Título: Desarrollo de una herramienta de pruebas para aplicaciones basadas en eventos que usan RabbitMQ

1. Énfasis: Ingeniería de Software
2. Área de trabajo: Aseguramiento de calidad, arquitecturas orientadas a eventos, pruebas automatizadas
3. Tipo de proyecto: Aplicado
4. Estudiante: Ever Daniel Rivera Inagán
5. Correo electrónico: edaniel@javerianacali.edu.co
6. Dirección y teléfono: Cra 42 23-116 - +57 318 245 0917
7. Director: Msc. Juan Pablo García Cifuentes
8. Vinculación del director: Planta
9. Correo electrónico del director: jpgarcia@javerianacali.edu.co
10. Palabras clave: Arquitecturas orientadas a eventos, RabbitMQ, pruebas automatizadas, trazabilidad de eventos, AsyncAPI, aseguramiento de calidad
11. ODS que aplica al proyecto (Agenda 2030): ODS 9 - Industria, innovación e infraestructura
12. Fecha de inicio: 01/07/2025
13. Resumen: Este proyecto aborda el desafío del aseguramiento de calidad en arquitecturas orientadas a eventos que utilizan RabbitMQ como intermediario de mensajes. La naturaleza asíncrona y no determinista de estos sistemas dificulta la trazabilidad de eventos, la detección de fallos intermitentes y la validación de contratos entre servicios. Se desarrolló una herramienta de pruebas automatizadas siguiendo la metodología Design Thinking, estructurada en un monolito modular que integra capacidades de monitoreo, validación de contratos mediante AsyncAPI y depuración en tiempo real. La validación se realizó mediante seis escenarios que evalúan violaciones de contrato, eventos perdidos, mensajes duplicados, orden incorrecto, configuración de infraestructura y eventos huérfanos.

Agradecimientos

Al finalizar este trabajo de grado, deseo expresar mi profundo agradecimiento a quienes hicieron posible su realización.

En primer lugar, agradezco al Msc. Juan Pablo García Cifuentes, director de este proyecto, por su guía experta, su disposición constante y su compromiso con la excelencia académica. Sus orientaciones fueron fundamentales para mantener el rigor metodológico y la claridad conceptual a lo largo de todo el proceso de investigación.

A mi pareja, quien me acompañó durante este intenso camino académico con paciencia, comprensión y apoyo incondicional. Su aliento en los momentos de mayor exigencia fue esencial para mantener la motivación y alcanzar este logro.

A la Pontificia Universidad Javeriana Cali y a la Facultad de Ingeniería y Ciencias, por brindarme los recursos, el espacio académico y las herramientas necesarias para desarrollar este proyecto de investigación.

A los profesionales que participaron en las entrevistas y encuestas, compartiendo generosamente su experiencia y conocimiento sobre arquitecturas orientadas a eventos. Sus aportes fueron invaluables para comprender las necesidades reales de la industria y orientar el desarrollo de la solución propuesta.

Finalmente, agradezco a todos quienes, de una u otra forma, contribuyeron a mi formación profesional y al desarrollo de esta investigación.

Resumen

En los últimos años, las arquitecturas orientadas a eventos (Event-Driven Architectures, EDA) han sido ampliamente adoptadas en el desarrollo de sistemas distribuidos debido a su capacidad para procesar grandes volúmenes de información, facilitar la comunicación asíncrona y mejorar la escalabilidad. Entre las tecnologías utilizadas para su implementación se encuentra RabbitMQ, un intermediario de mensajes que soporta protocolos como AMQP y MQTT. Sin embargo, la naturaleza asíncrona y no determinista de estas arquitecturas plantea retos significativos en el aseguramiento de calidad, particularmente en la trazabilidad de eventos y en la detección de fallos intermitentes. En este contexto, se diseñó e implementó una herramienta de pruebas automatizadas para sistemas basados en eventos que emplean RabbitMQ, integrando mecanismos de monitoreo, validación y depuración. Se utilizó el enfoque Design Thinking como metodología, siguiendo un ciclo iterativo conformado por las fases de empatizar, definir, idear, prototipar y testear. Se conceptualizó la solución, se desarrolló un prototipo funcional y se validó en un entorno de pruebas controlado. Se ejecutaron seis escenarios de evaluación que cubren violaciones de contrato, eventos perdidos, mensajes duplicados, orden incorrecto, errores de configuración de infraestructura y detección de eventos huérfanos. Los resultados demostraron una tasa de detección del 100 % de las anomalías introducidas, con reportes de alta precisión diagnóstica y observabilidad completa mediante integración con Grafana Loki. La herramienta reduce significativamente el tiempo de depuración, mejora la confiabilidad del sistema y facilita la integración en flujos de trabajo DevOps. El proyecto contribuye tanto académicamente mediante documentación de buenas prácticas como profesionalmente al ofrecer una solución viable y escalable para la industria.

Palabras Clave: Arquitecturas orientadas a eventos, RabbitMQ, pruebas automatizadas, trazabilidad de eventos, AsyncAPI, aseguramiento de calidad, sistemas distribuidos, mensajería asíncrona. proyecto.

Abstract

In recent years, Event-Driven Architectures (EDA) have gained widespread adoption in the development of distributed systems due to their ability to process large volumes of information, enable asynchronous communication, and improve scalability. Among the technologies used for their implementation is RabbitMQ, a message broker that supports protocols such as AMQP and MQTT. However, the asynchronous and non-deterministic nature of these architectures poses significant challenges in quality assurance, particularly in event traceability and intermittent failure detection. In this context, an automated testing tool was designed and implemented for event-based systems using RabbitMQ, integrating monitoring, validation, and debugging mechanisms. The Design Thinking approach was applied as a methodology, following an iterative cycle consisting of empathize, define, ideate, prototype, and test phases. The solution was conceptualized, a functional prototype was developed, and it was validated in a controlled testing environment. Six evaluation scenarios were executed covering contract violations, lost events, duplicate messages, incorrect order, infrastructure configuration errors, and orphan events detection. Results demonstrated a 100 % detection rate of introduced anomalies, with high diagnostic precision reports and complete observability through integration with Grafana Loki. The tool significantly reduces debugging time, improves system reliability, and facilitates integration into DevOps workflows. The project contributes both academically through documentation of best practices and professionally by offering a viable and scalable solution for the industry.

Keywords: Event-driven architecture, RabbitMQ, automated testing, event traceability, AsyncAPI, quality assurance, distributed systems, asynchronous messaging.

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En los últimos años, las arquitecturas orientadas a eventos (Event-Driven Architectures, EDA) han sido adoptadas en el desarrollo de sistemas distribuidos debido a su capacidad para procesar grandes volúmenes de información, facilitar la comunicación asíncrona y mejorar la escalabilidad (Rahmatulloh et al. (2024)). Entre las tecnologías utilizadas para su implementación se encuentra RabbitMQ, un intermediario de mensajes que soporta protocolos como AMQP y MQTT (TheirStack (sf)). Sin embargo, la naturaleza asíncrona y no determinista de estas arquitecturas plantea retos significativos en el aseguramiento de calidad, particularmente en la trazabilidad de eventos y en la detección de fallos intermitentes (Wu et al. (2022); Herbold and Harms (2013)). En este contexto, se diseñó e implementó una herramienta de pruebas automatizadas para sistemas basados en eventos que emplean RabbitMQ, la cual integró mecanismos de monitoreo, validación y depuración. Esta solución tuvo como propósito apoyar la labor de los equipos de desarrollo y aseguramiento de calidad, al facilitar la detección temprana de errores, la verificación de contratos entre servicios y la evaluación de flujos de eventos en entornos distribuidos.

Se utilizó el enfoque de Design Thinking como metodología, siguiendo un ciclo iterativo conformado por las fases de empatizar, definir, idear, prototipar y testear (Leifer et al. (2020)). Durante este proceso se conceptualizó la solución, se desarrolló un prototipo funcional y se validó en un entorno de pruebas controlado.

La herramienta implementada tuvo relevancia tanto académica como práctica. Desde el ámbito académico, aportó conocimiento y buenas prácticas documentadas para el aseguramiento de calidad en arquitecturas orientadas a eventos. En el contexto profesional, contribuyó a mejorar la calidad del software mediante una mayor trazabilidad de los eventos, reducción de tiempos de depuración y optimización del uso de recursos, generando un impacto positivo en la confiabilidad y eficiencia operativa (Consortium for Information & Software Quality (CISQ) (2022); Datadog (2022)).

El presente documento se estructura de la siguiente manera: en este primer capítulo se presenta la introducción general, que incluye el contexto, los objetivos, la metodología y la relevancia del trabajo. En el segundo capítulo se expone el marco teórico y el estado del arte. El tercer capítulo describe la metodología aplicada y el desarrollo de la herramienta. En el cuarto capítulo se presentan los resultados y la validación. Finalmente, el quinto capítulo incluye las conclusiones y el trabajo futuro.

1.1. Definición del problema

En los sistemas distribuidos que emplean arquitecturas orientadas a eventos y RabbitMQ como intermediario de mensajes, se evidenció la ausencia de mecanismos nativos que permitieran rastrear el flujo de los eventos, identificar fallos intermitentes y verificar la coherencia en la comunicación entre servicios. Esta limitación representa un desafío central para el aseguramiento de calidad, al incrementar la complejidad de las pruebas y elevar el riesgo de que defectos no fueran detectados oportunamente, generando impactos directos en la operación y en los costos de mantenimiento (Consortium for Information & Software Quality (CISQ) (2022)).

1.1.1. Planteamiento del problema

La naturaleza asíncrona de los sistemas basados en eventos introdujo incertidumbre en el orden y en los tiempos de entrega de los mensajes, lo que dificulta la trazabilidad y la detección de errores intermitentes. En entornos que utilizan intermediarios de mensajes, esta situación se vio agravada por la falta de mecanismos especializados para observar y validar el comportamiento de flujos complejos de eventos.

De mantenerse estas limitaciones, las organizaciones se encontrarían expuestas a pérdidas económicas significativas, a un deterioro progresivo de la calidad del software y a afectaciones en su reputación, comprometiendo con ello su competitividad en el mercado (Consortium for Information & Software Quality (CISQ) (2022)).

1.1.2. Formulación del problema

¿Cómo desarrollar e implementar una herramienta de pruebas automatizadas que permita monitorear, validar y depurar flujos en arquitecturas orientadas a eventos basadas en RabbitMQ, con el fin de mejorar la trazabilidad, la detección temprana de errores y la confiabilidad del sistema?

1.1.3. Sistematización

A partir del problema identificado, se plantearon las siguientes preguntas de investigación que orientaron el desarrollo e implementación de la herramienta:

- ¿Qué limitaciones presentan las metodologías y herramientas de pruebas existentes al aplicarse en arquitecturas orientadas a eventos?
- ¿Qué mecanismos de monitoreo, validación y depuración pueden implementarse para mejorar la trazabilidad y la detección temprana de errores en flujos de eventos asíncronos?

- ¿Cómo evaluar de manera objetiva la efectividad de una herramienta de pruebas en arquitecturas orientadas a eventos, considerando criterios de cobertura, detección de errores, confiabilidad y facilidad de uso?

Estas preguntas guiaron la investigación, orientando tanto el diseño conceptual como el desarrollo metodológico y la validación experimental de la solución propuesta.

1.2. Objetivos del proyecto

1.2.1. Objetivo General

Desarrollar una herramienta de pruebas automatizadas para arquitecturas orientadas a eventos basadas en RabbitMQ, que integre capacidades de monitoreo, validación y depuración, con el propósito de mejorar la trazabilidad de los eventos y la detección temprana de errores en entornos distribuidos.

1.2.2. Objetivos específicos

- Analizar las limitaciones de las metodologías y herramientas de pruebas existentes aplicadas a arquitecturas orientadas a eventos.
- Diseñar la arquitectura y los componentes de una herramienta de pruebas automatizadas que integre mecanismos de trazabilidad, validación y depuración de eventos.
- Implementar la herramienta propuesta siguiendo principios de ingeniería de software y buenas prácticas de desarrollo para entornos distribuidos asíncronos, orientada a mejorar la calidad del proceso de pruebas.
- Definir métricas e indicadores que permitan evaluar la efectividad de la herramienta en términos de cobertura, confiabilidad y facilidad de uso.
- Validar la herramienta mediante pruebas funcionales y no funcionales en un entorno controlado que reproduzca escenarios reales de operación.

1.3. Delimitaciones y alcances

- El desarrollo de este trabajo se centró exclusivamente en la creación de una herramienta de pruebas automatizadas para sistemas basados en eventos que utilizan RabbitMQ. La investigación y la implementación se limitaron a entornos de arquitectura orientada a eventos, sin abordar otros paradigmas de comunicación, como las arquitecturas monolíticas o cliente-servidor tradicionales.

- Para la experimentación, se emplearon escenarios simulados que representaron situaciones comunes en sistemas distribuidos asíncronos, sin incluir pruebas en entornos de producción.
- El diseño de la arquitectura de la herramienta se elaboró a partir de los requerimientos identificados durante la etapa de empatizar, descartando características no esenciales que pudieran ampliar significativamente el alcance o los tiempos de desarrollo.
- La validación de la herramienta se llevó a cabo mediante pruebas funcionales y no funcionales orientadas a evaluar su cobertura, capacidad de detección de errores, confiabilidad y facilidad de uso. No se abordó la integración con sistemas de monitoreo comerciales ni la automatización de despliegues en la nube, los cuales se consideran líneas de trabajo futuras.

En síntesis, el alcance del proyecto comprendió la entrega de una herramienta funcional validada en entornos controlados, acompañada de la documentación técnica necesaria para su uso y con una base sólida que facilite futuras ampliaciones o integraciones en contextos más complejos.

1.4. Justificación del trabajo

La herramienta desarrollada respondió a una necesidad concreta en el aseguramiento de calidad de sistemas distribuidos basados en arquitecturas orientadas a eventos. Al resolver las limitaciones identificadas en la trazabilidad, validación y depuración de flujos, la solución permitió reducir la complejidad y el tiempo necesario para detectar y corregir errores, lo que impactó positivamente en la eficiencia operativa y en la calidad final del software (Wu et al. (2022); Herbold and Harms (2013)).

En términos económicos, la reducción de los tiempos de depuración y la detección temprana de defectos contribuyeron a disminuir los costos asociados al mantenimiento correctivo y a la atención de incidentes en producción. De acuerdo con el Consortium for Information & Software Quality (CISQ) (2022), los defectos de software no detectados oportunamente pueden representar pérdidas multimillonarias para las organizaciones. De forma complementaria, la investigación de Boehm and Basili (2001) evidenció que el costo de corregir un defecto se incrementa de manera exponencial cuanto más avanzada se encuentre la etapa del ciclo de desarrollo en la que es identificado.

Desde el punto de vista técnico, la herramienta constituyó una base reutilizable para validar de forma automatizada los flujos de eventos en entornos asíncronos, proporcionando un enfoque metodológico adaptable a distintos escenarios de la industria. Esta capacidad

resulta clave, dado que la trazabilidad de eventos es un factor crítico para garantizar la confiabilidad en arquitecturas orientadas a eventos (Rahmatulloh et al. (2024); Lee (2009)). Además, la propuesta contribuyó a reducir la brecha existente en la disponibilidad de soluciones código abierto específicas para RabbitMQ, un campo en el que la literatura técnica evidencia carencias y oportunidades de innovación. (Wu et al. (2022)).

En el ámbito académico, el trabajo fortaleció el conocimiento en ingeniería de software aplicada a entornos de mensajería asíncrona, generando documentación técnica y evidencia empírica que pueden servir de base para investigaciones posteriores. La relevancia de este tipo de aportes radica en que las arquitecturas orientadas a eventos continúan expandiéndose en sectores como el comercio electrónico, el Internet de las Cosas (IoT) y el procesamiento en tiempo real (Silva et al. (2025)), lo que incrementa la necesidad de contar con métodos robustos de validación.

En síntesis, la solución desarrollada permitió mejorar significativamente las condiciones que enfrentaban los equipos de desarrollo en la validación de sistemas asíncronos, generando beneficios medibles en costos, calidad del software, confiabilidad operativa y transferencia de conocimiento hacia la comunidad técnica y académica.

1.5. Metodología de la investigación

La investigación se desarrolló siguiendo el enfoque Design Thinking, seleccionado por su capacidad para abordar problemas complejos de manera iterativa, centrada en el usuario y orientada a la generación de soluciones prácticas (Leifer et al. (2020)). Esta metodología permitió alinear el proceso de implementación de la herramienta con las necesidades reales de los equipos de desarrollo y aseguramiento de calidad, garantizando que cada fase respondiera a los objetivos planteados.

El proceso metodológico se estructuró en cinco fases principales:

- **Empatizar:** Se recopilaron datos mediante revisión de literatura, análisis de documentación técnica, encuestas y entrevistas semiestructuradas a profesionales con experiencia en arquitecturas orientadas a eventos. Esta fase permitió identificar limitaciones concretas en la trazabilidad, validación y depuración de eventos.
- **Definir:** Se organizaron y sistematizaron los hallazgos para formular un enunciado claro del problema y priorizar los requisitos de la solución.
- **Idear:** Se generó una propuesta de solución, evaluando su viabilidad técnica y su alineación con los objetivos planteados. Se seleccionó un diseño de arquitectura modular para la herramienta, considerando su facilidad de integración y mantenimiento.

- **Prototipar:** Se desarrolló un prototipo funcional en un entorno controlado, incorporando capacidades de monitoreo, validación y depuración de flujos de eventos. La implementación se llevó a cabo siguiendo principios de ingeniería de software y buenas prácticas en el desarrollo backend.
- **Testear:** El prototipo fue evaluado mediante pruebas funcionales y no funcionales orientadas a medir cobertura, detección de errores, confiabilidad y facilidad de uso. Los resultados obtenidos se analizaron para validar el cumplimiento de los objetivos y detectar oportunidades de mejora.

La información recolectada en cada fase fue organizada y documentada para facilitar su interpretación y presentación. El análisis de los datos combinó técnicas cualitativas, derivadas de las entrevistas y de la revisión de literatura, con métricas cuantitativas obtenidas de las pruebas realizadas. Esta combinación permitió no solo validar la efectividad de la herramienta, sino también aportar evidencia empírica sobre su impacto en la mejora de los procesos de aseguramiento de calidad en sistemas asíncronos.

1.6. Resultados obtenidos

El desarrollo e implementación de la herramienta de pruebas automatizadas para arquitecturas orientadas a eventos basadas en RabbitMQ generó resultados significativos, al validar la efectividad de la solución propuesta para abordar las limitaciones identificadas en el aseguramiento de calidad de sistemas distribuidos asíncronos.

Se implementó un prototipo funcional estructurado bajo un enfoque monolítico modular que integra capacidades de monitoreo, validación de contratos mediante especificaciones AsyncAPI y depuración en tiempo real. La arquitectura modular facilita la extensibilidad y el mantenimiento, permitiendo agregar nuevos validadores sin alterar componentes existentes.

En términos de efectividad operacional, la evaluación mediante seis escenarios representativos de problemas recurrentes en sistemas distribuidos demostró una tasa de detección del 100 % de las anomalías introducidas, sin generar falsos negativos.

La precisión de diagnóstico alcanzó consistentemente un nivel alto en todos los escenarios, proporcionando reportes estructurados con información contextual detallada. Esto permite a los desarrolladores comprender la naturaleza exacta del problema, identificar el componente específico que debe corregirse y reproducir localmente las condiciones del fallo sin necesidad de investigación exploratoria adicional.

Capítulo 2

Marco de referencia

2.1. Marco Teórico

2.1.1. Arquitectura orientada a eventos (EDA)

La Arquitectura Orientada a Eventos (Event-Driven Architecture, EDA) es un estilo arquitectónico ampliamente adoptado en el desarrollo de sistemas distribuidos y descentralizados. En este enfoque, los componentes intercambian información de manera asíncrona mediante la publicación y el consumo de eventos gestionados por mecanismos de mensajería. Este patrón permite reducir el acoplamiento entre servicios, aumentar la escalabilidad y mejorar el rendimiento a través del procesamiento asíncrono(Rahmatulloh et al. (2024)).

Una de las implementaciones más comunes de este modelo emplea intermediarios de mensajes como Apache Kafka, RabbitMQ o ActiveMQ, los cuales gestionan colas, enrutan mensajes y garantizan la entrega conforme a distintas políticas de persistencia y confirmación.

2.1.2. Teorema CAP

El teorema CAP, demostrado formalmente por Gilbert and Lynch (2002), establece que en sistemas distribuidos, establece que en sistemas distribuidos es imposible garantizar simultáneamente tres propiedades: Consistencia (C), Disponibilidad (A) y Tolerancia a particiones (P). En presencia de una partición de red, un sistema debe elegir entre mantener la consistencia o la disponibilidad.

- **Consistencia:** Todos los nodos del sistema observan los mismos datos al mismo tiempo. Una lectura siempre retorna el resultado de la escritura más reciente.
- **Disponibilidad:** Cada petición recibe una respuesta, sin garantía de que contenga la versión más reciente de la información. El sistema permanece operativo incluso cuando algunos nodos fallan.
- **Tolerancia a particiones:** El sistema continúa operando a pesar de que se pierda conectividad de red entre algunos nodos, generando particiones en el clúster.

Según Kleppmann (2017), en sistemas de mensajería distribuidos, la tolerancia a particiones no es opcional: las fallas de red son inevitables en entornos distribuidos. Por tanto, el diseño debe elegir entre consistencia (CP) o disponibilidad (AP).

2.1.2.1. Modos de operación de RabbitMQ según el teorema CAP

RabbitMQ implementa diferentes estrategias de replicación que adoptan posiciones distintas frente al teorema CAP, según la configuración de colas y la topología del clúster.

Colas de quórum: Implementan el algoritmo de consenso Raft (Ongaro and Ousterhout (2014)), garantizando que las escrituras solo se confirman cuando una mayoría de réplicas las ha persistido. Ante particiones de red que impidan alcanzar *quórum*, las escrituras se bloquean, priorizando consistencia sobre disponibilidad. RabbitMQ (2024b)

***Colas Stream (AP)*:** Emplean replicación asíncrona, permitiendo que el líder continúe aceptando escrituras aunque las réplicas presenten lag. Priorizan disponibilidad sobre consistencia estricta. RabbitMQ (2024c)

Colas clásicas con réplicas espejo (CP): Replican mensajes de forma síncrona entre nodos. Durante el failover, la cola permanece temporalmente inaccesible, manteniendo un enfoque CP. RabbitMQ (2024a)

2.1.3. Pruebas en microservicios orientados a eventos

A diferencia de los microservicios basados en REST (Representational State Transfer), donde las pruebas se apoyan en técnicas consolidadas como las pruebas unitarias, de integración o punto a punto, los entornos EDA presentan retos adicionales derivados de la comunicación asíncrona y la ejecución distribuida. Esta condición dificulta la trazabilidad, introduce condiciones de carrera y complica la reproducibilidad de errores en ambientes altamente dispersos.

Entre los enfoques propuestos para abordar estas limitaciones se encuentran:

- **Pruebas de contratos dirigidas por el consumidor:** la técnica Consumer-Driven Contract Testing (CDCT) valida la comunicación entre productor y consumidor de mensajes a partir de contratos previamente definidos, garantizando que las interacciones cumplan con el formato y comportamiento acordados (Wu et al. (2022)).
- **Modelos de estado y patrón SAGA:** representan formalmente las transiciones de estado en flujos de larga duración, lo que permite detectar estados aislados, rutas incompletas o bucles infinitos (Wu et al. (2022)).

- **Monitoreo distribuido:** se basa en técnicas de rastreo distribuido que permiten seguir eventos a lo largo de múltiples servicios y facilitar su diagnóstico (Wu et al. (2022)).

En este contexto, diversos estudios resaltan que la naturaleza asíncrona y el alto volumen de datos procesados en las arquitecturas EDA exigen estrategias de prueba que no solo verifiquen la funcionalidad, sino que también evalúen la consistencia de datos, la resiliencia y la tolerancia a fallos. Esto implica la integración de herramientas capaces de realizar seguimiento de eventos en tiempo real y validaciones de estado en sistemas distribuidos, especialmente en escenarios de alta concurrencia y escalabilidad (Silva et al. (2025))

2.1.4. Plataformas y marcos de prueba para EDA

La literatura especializada propone diversos marcos que buscan desacoplar las técnicas de prueba de la plataforma tecnológica específica, facilitando la validación de sistemas orientados a eventos bajo distintos entornos de ejecución:

- **AutoQUEST:** es un marco de pruebas que abstrae la captura, generación y ejecución de pruebas para software orientado a eventos mediante una API de eventos genérica y el uso de complementos adaptables a diferentes plataformas (Herbold and Harms (2013)). Integra enfoques de pruebas basadas en uso, análisis de usabilidad y técnicas de captura y reproducción, promoviendo la reutilización y la extensibilidad.
- **BOSET:** es una plataforma de pruebas que incorpora simulación de procesos de negocio (Business Process Automation, BPA) para evaluar sistemas SOA y orientados a eventos. Permite analizar la robustez y el desempeño del sistema en escenarios complejos y con múltiples interacciones (Lee (2009)).

2.1.5. Calidad estructural del software

El Consortium for Information & Software Quality (CISQ) (2022) ha estandarizado un conjunto de métricas automatizables para evaluar atributos estructurales del software como la seguridad, la confiabilidad, la mantenibilidad y la eficiencia del desempeño a nivel de código fuente. Estas métricas, alineadas con el marco ISO/IEC 25000 y sus derivados, permiten detectar violaciones críticas a las buenas prácticas de arquitectura y codificación. Tales aspectos resultan especialmente relevantes en entornos EDA, donde una falla en un servicio puede propagarse y comprometer el funcionamiento de todo el sistema.

2.1.6. Costos y riesgos asociados a defectos

De acuerdo con Boehm and Basili (2001), encontrar y corregir un defecto después de la entrega de un sistema puede resultar hasta cien veces más costoso que hacerlo durante las

fases tempranas del desarrollo. En el caso de los sistemas distribuidos basados en eventos, este costo potencial se incrementa debido a la dificultad para reproducir y aislar fallos asociados a la comunicación asíncrona.

La aplicación de prácticas como revisiones por pares, pruebas basadas en riesgo y procesos de desarrollo disciplinados permite reducir hasta en un 75 % la introducción de errores, mitigando así los riesgos económicos y operativos derivados de defectos no detectados oportunamente.

2.2. Estado del Arte

El análisis de trabajos previos evidencia la existencia de diferentes enfoques para la realización de pruebas en arquitecturas orientadas a eventos, con el propósito de identificar avances, limitaciones y oportunidades de mejora en el aseguramiento de calidad de sistemas distribuidos y asíncronos. A continuación se presenta un resumen comparativo de los principales aportes encontrados en la literatura:

Tabla 1: Resumen de trabajos previos sobre pruebas en arquitecturas orientadas a eventos

Referencia / Propuesta	**Enfoque principal**	**Ventajas**	**Limitaciones**
CCTS (Composite Contract Testing Service) Wu et al. (2022)	Integra pruebas de contrato dirigidas por el consumidor con modelos de estado para verificar flujos SAGA y contratos de mensajería.	Detecta estados aislados, bucles infinitos y secuencias inválidas.	Requiere instrumentar los servicios con metadatos específicos.
BOSET Lee (2009)	Marco de pruebas basado en simulación de procesos de negocio para entornos SOA y EDA.	Evalúa robustez y rendimiento en escenarios complejos.	Su enfoque original está orientado a SOA, por lo que requiere adaptación para arquitecturas EDA.

AutoQUEST Herbold and Harms (2013)	Plataforma que abstrae la captura, generación y ejecución de pruebas mediante una API genérica de eventos y el uso de complementos para distintas plataformas.	Favorece la independencia de plataforma y soporta pruebas basadas en uso, captura y reproducción, y análisis de usabilidad.	Presenta complejidad inicial en la creación de complementos específicos.

Capítulo 3

Desarrollo del Proyecto

Este capítulo describe de manera integral el proceso de desarrollo de la herramienta de pruebas automatizadas para arquitecturas basadas en eventos que utilizan RabbitMQ. En él se narran las actividades prácticas realizadas desde la fase inicial de exploración hasta la implementación y validación del prototipo, siguiendo los principios de la metodología Design Thinking.

El contenido se organiza conforme a las fases de dicha metodología (empatizar, definir, idear, prototipar y testear), lo que permite mostrar de forma secuencial cómo los hallazgos obtenidos en las entrevistas, encuestas, análisis documental y experiencias profesionales se transformaron progresivamente en requisitos, diseños y, finalmente, en un sistema funcional. Asimismo, se presentan los resultados obtenidos en cada fase, los mecanismos de validación aplicados y la forma en que se identificaron y abordaron los principales retos técnicos y metodológicos del proyecto.

3.1. Fase de Empatizar

La primera fase de la metodología Design Thinking correspondió a la etapa de empatizar, cuyo propósito fue comprender en profundidad el contexto del problema y las experiencias de los actores involucrados. Esta fase buscó identificar las principales dificultades que enfrentan los equipos de desarrollo y aseguramiento de calidad en la validación de aplicaciones basadas en arquitecturas orientadas a eventos.
Para ello, se definió un conjunto de pasos e instrumentos de recolección de información directa, cuyo propósito fue generar insumos que orientaran las fases posteriores de definición, ideación y prototipado. A continuación, se describe el proceso seguido en cada una de las actividades desarrolladas durante esta fase.

3.1.1. Identificación de actores y contexto

La fase de empatizar inició con la identificación de los actores clave involucrados en el proceso de desarrollo y aseguramiento de calidad de sistemas basados en arquitecturas orientadas a eventos. Esta etapa permitió delimitar el contexto donde se presentan las dificultades relacionadas con la trazabilidad, validación y depuración de flujos asíncronos. El análisis del contexto se centró en organizaciones que utilizan arquitecturas EDA. En este tipo de entornos, los sistemas se componen de múltiples servicios que publican y consumen mensajes de manera asíncrona, lo que introduce complejidades en la observación de los flujos, la detección de errores y la sincronización de estados. Estas condiciones fueron determinantes para comprender la naturaleza del problema que la herramienta debía abordar.
A partir de esta caracterización, se identificaron tres grupos de actores principales:

- **Desarrolladores backend:** responsables de implementar los servicios que publican o consumen eventos en RabbitMQ. Su interés radica en contar con mecanismos de prueba que les permitan validar el comportamiento de los flujos sin depender del entorno productivo.
- **Ingenieros de aseguramiento de calidad (QA):** encargados de diseñar, ejecutar y automatizar las pruebas funcionales y no funcionales. En el contexto EDA, enfrentan limitaciones para reproducir condiciones de fallo y verificar la trazabilidad completa de los mensajes.
- **Arquitectos de software:** supervisan la coherencia de la arquitectura distribuida, definen políticas de comunicación entre servicios y validan la conformidad de los contratos. Su principal desafío consiste en garantizar la confiabilidad y consistencia del sistema a medida que aumenta la complejidad de los flujos de eventos.

La comprensión del rol y las necesidades de estos actores resultó esencial para orientar la recolección de información y definir los criterios de diseño de la herramienta.

3.1.2. Técnicas de recolección de información

Durante la fase de empatizar se empleó una combinación de técnicas cualitativas y cuantitativas con el propósito de comprender de manera integral las necesidades, dificultades y expectativas de los actores identificados. Se utilizaron tres instrumentos principales: entrevistas semi-estructuradas, encuestas en línea y análisis de documentación técnica.

3.1.2.1. Entrevistas semi-estructuradas

Con el fin de profundizar en las prácticas actuales de prueba en sistemas asíncronos, se diseñó una guía de entrevista semiestructurada orientada a explorar las experiencias, retos y expectativas de profesionales que trabajan con arquitecturas orientadas a eventos. Las entrevistas se estructuraron a partir de un conjunto de preguntas abiertas enfocadas en cuatro ejes principales: prácticas en pruebas, herramientas utilizadas, retos de trazabilidad y características esperadas en una herramienta ideal. La guía completa de preguntas puede consultarse en el Anexo A.

El proceso de diseño de la guía se fundamentó en buenas prácticas de entrevistas cualitativas, promoviendo la escucha activa, evitando preguntas dirigidas y favoreciendo la narración libre por parte de los participantes. Para ello, se consideraron las recomendaciones metodológicas de Headlee (2015), quien resalta la importancia de fomentar conversaciones auténticas y profundas, y de Babich (2019), quien enfatiza la claridad en los objetivos y la formulación de preguntas abiertas en entrevistas con usuarios.

Las entrevistas se realizaron de manera virtual, con una duración promedio de 30 minutos cada una. Se obtuvo consentimiento informado de los participantes para el uso académico de sus respuestas y se tomaron notas detalladas que posteriormente fueron analizadas y codificadas para identificar temas recurrentes y hallazgos relevantes.

3.1.2.2. Encuestas

Con el propósito de complementar la información obtenida en las entrevistas y observar patrones comunes a nivel cuantitativo, se diseñó y aplicó una encuesta en línea mediante la plataforma Microsoft Forms. El formulario fue compartido con desarrolladores, ingenieros de aseguramiento de calidad, arquitectos de software y otros profesionales del gremio relacionados con la validación de sistemas basados en eventos.

La encuesta incluyó preguntas cerradas y abiertas orientadas a identificar el rol de los participantes, su nivel de experiencia con arquitecturas EDA y RabbitMQ, la frecuencia con la que enfrentan problemas difíciles de depurar y la percepción de confiabilidad respecto

a las herramientas que emplean actualmente. De manera complementaria, se indagó sobre la ocurrencia de fallos en producción, las dificultades específicas en la validación de sistemas asíncronos y las herramientas utilizadas en su práctica profesional.
El diseño del cuestionario buscó equilibrar preguntas de opción múltiple, que facilitaran el análisis comparativo, con preguntas abiertas, que permitieran capturar percepciones y necesidades no contempladas inicialmente. Esta combinación es recomendada en la literatura sobre levantamiento de requerimientos y metodologías aplicadas, ya que posibilita identificar tanto tendencias cuantitativas como matices cualitativos relevantes para el diseño de soluciones de software (Koziol and Bilder (2014); Headlee (2015)).
La encuesta completa se incluye en el Anexo B para su consulta detallada. Los resultados obtenidos se integraron con los hallazgos de las entrevistas, aportando evidencia empírica para priorizar los requisitos funcionales de la herramienta propuesta.

3.1.2.3. Documentación

De manera complementaria al trabajo de campo, se revisó la documentación técnica y la literatura aplicada sobre herramientas de pruebas orientadas a arquitecturas basadas en eventos. El propósito de este análisis fue identificar qué soluciones concretas se encuentran disponibles, cómo se describen en sus manuales y reportes, y cuáles son sus principales características desde la perspectiva de los equipos de desarrollo que trabajan con mensajería asíncrona.
En este ejercicio se analizaron las propuestas académicas presentadas en la Tabla 1, donde se observó que la mayoría corresponden a prototipos funcionales orientados a contextos de investigación o pruebas controladas. De acuerdo con sus publicaciones, estas herramientas ofrecen mecanismos de validación valiosos, pero presentan baja adopción industrial, atribuida principalmente a la complejidad de despliegue, la necesidad de instrumentación manual y la falta de integración con flujos DevOps. Estos aspectos se reconocieron como condiciones relevantes del contexto actual de las pruebas en sistemas basados en eventos.
Por otra parte, se revisaron herramientas de amplio uso en la industria, como Postman y JMeter, ampliamente documentadas y adoptadas para la validación de APIs REST, pruebas de rendimiento y generación de cargas de trabajo (Postman (2025); Apache Software Foundation (2025)). A partir de su documentación oficial y casos de uso reportados, se constató que ambas se enfocan en arquitecturas síncronas basadas en peticiones HTTP, lo que limita su aplicación directa en escenarios EDA.
Este análisis documental permitió contextualizar el panorama actual de herramientas de prueba, evidenciando tanto los aportes de la literatura académica como las brechas que persisten en las soluciones de uso industrial. Estos hallazgos sirvieron como base para orientar la formulación del problema en la siguiente fase del proceso metodológico.

3.1.3. Resumen de la fase

La fase de empatizar permitió obtener una comprensión integral de las dificultades que enfrentan los equipos de desarrollo y aseguramiento de calidad en la validación de sistemas basados en eventos. A partir de la interacción con los profesionales y del análisis de herramientas existentes, se identificaron carencias en la trazabilidad de eventos, la depuración de flujos asíncronos y la ausencia de soluciones específicas para RabbitMQ. Esta información constituyó el insumo principal para la fase siguiente, en la cual se definieron los requerimientos y criterios de diseño de la herramienta propuesta.

3.2. Fase de Definir

La segunda fase de la metodología Design Thinking estuvo orientada a definir el problema y los requerimientos de la solución, a partir de los resultados obtenidos en la etapa de empatizar. En esta fase se buscó transformar los hallazgos iniciales en un planteamiento claro del problema central, identificando las causas principales, los efectos observados y las oportunidades de mejora dentro del proceso de pruebas de aplicaciones orientadas a eventos.

Para ello, en los apartados siguientes se organizaron, analizaron y clasificaron los datos recolectados en las entrevistas, encuestas y revisión documental, complementados con observaciones directas sobre el entorno de validación de sistemas asíncronos. Este proceso permitió identificar patrones de comportamiento, necesidades recurrentes y limitaciones comunes en las herramientas y prácticas actuales en las pruebas lo que orientó las fases posteriores de ideación y prototipado.

3.2.1. Análisis de las entrevistas

3.2.1.1. Técnicas de análisis

Para el análisis de las transcripciones de las entrevistas se adoptó una estrategia híbrida, combinando métodos de análisis semántico y técnicas de agrupamiento no supervisado, en concordancia con las prácticas recomendadas en la literatura especializada.

En primer lugar, se generaron representaciones semánticas de las oraciones mediante Sentence-BERT (SBERT), un modelo que extiende BERT para producir vectores de alta dimensionalidad capaces de capturar la similitud semántica entre textos. Esta técnica permitió representar el contenido de las entrevistas con una comprensión contextual más allá del conteo de palabras, mejorando significativamente la eficacia del agrupamiento en textos extensos (Reimers and Gurevych (2019)).

Posteriormente, se aplicaron técnicas de agrupamiento no supervisado, específicamente K-Means y HDBSCAN. El algoritmo K-Means se utilizó para explorar agrupamientos iniciales y determinar la densidad de los datos, mientras que HDBSCAN resultó especialmente valioso para detectar automáticamente clústeres temáticos sin requerir la especificación previa del número de grupos. Este enfoque fue apropiado en un contexto exploratorio como el de las entrevistas cualitativas, donde los patrones emergen de forma no estructurada (Malzer and Baum (2019)).

En la Tabla 2 se resumen las principales técnicas empleadas durante el análisis de las entrevistas, junto con una breve descripción y los beneficios que aportaron al proceso. Estas estrategias se combinaron para fortalecer la consistencia del análisis y mejorar la identificación de temas recurrentes en los datos textuales.

Tabla 2: Técnicas y beneficios del análisis aplicado a las entrevistas cualitativas

Técnica	Descripción	Beneficios
Sentence-BERT	Modelo basado en BERT que genera representaciones vectoriales de oraciones preservando su significado semántico.	Permite capturar similitudes contextuales más allá de coincidencias literales de palabras, mejorando la eficacia del agrupamiento en textos extensos (Reimers and Gurevych (2019)).
K-Means	Algoritmo de agrupamiento no supervisado que organiza datos en un número predefinido de clústeres.	Facilita la identificación de patrones generales en grandes volúmenes de texto y permite contrastar la coherencia entre grupos temáticos (Sampath and Vignesh (2024)).
HDBSCAN	Algoritmo de agrupamiento jerárquico basado en densidad que no requiere fijar el número de clústeres de antemano.	Detecta automáticamente agrupaciones temáticas y maneja datos con densidades variables, reduciendo el riesgo de forzar categorías artificiales (Malzer and Baum (2019)).
TF-IDF	Técnica de ponderación de términos que evalúa la importancia de una palabra dentro de un conjunto de documentos.	Mejora la interpretabilidad de los clústeres al identificar las palabras más representativas de cada grupo temático (Sampath and Vignesh (2024)).

3.2.1.2. Análisis

El análisis de las transcripciones se realizó con el apoyo de inteligencia artificial (IA), utilizando un conjunto de instrucciones específicas (prompt) que orientó el proceso de clasificación y síntesis de la información (ver Anexo C). Este enfoque permitió estructurar de manera sistemática las respuestas, reduciendo el sesgo interpretativo y favoreciendo la detección objetiva de patrones semánticos.

Mediante este procedimiento, las declaraciones de los participantes fueron organizadas en clústeres temáticos, lo que facilitó la identificación de tendencias comunes y necesidades recurrentes entre los profesionales entrevistados. En la Tabla 3 se presentan las principales

categorías emergentes del análisis, junto con las palabras asociadas y citas representativas que ilustran cada tema.

Tabla 3: Principales temas emergentes del análisis de entrevistas

C	Tema	Palabras	Citas representativas
1	**Trazabilidad y observabilidad de eventos**	flujo, consumidor, rastrear, error, logs, métricas	"Si algo falla en el consumidor de otro servicio… no hay manera sencilla de rehacer todo el flujo y rastrear el error". "Una herramienta de pruebas ideal debería dar métricas de latencia y trazabilidad centralizada". "Nosotros en Yuno tenemos que seguir el evento desde la pasarela hasta la conciliación y no hay una forma unificada de verlo". "Para Mars McLennan el problema es seguir los mensajes entre integraciones; los logs se pierden entre microservicios".
2	**Limitaciones de frameworks de prueba y baja automatización**	frameworks, unitarias, integración, Jest, automatización, simuladores, orquestación	"No existen frameworks que validen de extremo a extremo los escenarios asincrónicos de RabbitMQ". "Hace falta algo que orqueste casos end-to-end sobre los topics". "En Yuno los test automatizados se detienen en la API, no cubren los eventos; la validación sigue siendo manual". "En Ayesa las pruebas dependen de scripts aislados; no hay integración con los flujos de mensajería".

3	**Errores intermitentes y fallas en condiciones reales**	duplicidad, colas, intermitentes, reintentos, carga, delay	"En RabbitMQ es común que aparezcan duplicidades de mensajes… eso en pruebas controladas nunca se ve". "Sin una herramienta que permita simular cargas reales, es casi imposible anticipar errores". "En Yuno detectamos errores de concurrencia que solo aparecen en picos de transacciones reales". "En Antares los reintentos fallidos saturan la cola y generan mensajes huérfanos".
4	**Gestión de ambientes y datos de prueba**	ambientes, test, producción, simular, flujo, reproducibilidad	"En ambientes de test con respecto a producción… yo nada más he probado una vez en ese ambiente". "Esos ambientes son muy malos porque los devs tienden a romper cosas cuando comparten entornos de prueba". "Yuno tiene ambientes aislados, pero los datos de prueba no representan escenarios reales de pago; eso limita la cobertura". "En Mercado Libre la sincronización de ambientes entre colas y APIs es un reto constante".
5	**Requerimientos funcionales para una herramienta ideal de pruebas EDA**	contratos, payload, routing key, métricas, interfaz gráfica, AsyncAPI	"Me gustaría que validara los contratos, el payload, el routing key… y que mostrara gráficamente el flujo". "La herramienta que me gustaría tener es la de validar los contratos de eventos y verificar compatibilidad de esquemas". "Desde Yuno sería clave una interfaz que permita ver las trazas y validar los contratos con AsyncAPI". "Para Mars McLennan sería útil correlacionar los mensajes y generar alertas visuales en tiempo real".

6	**Rendimiento y capacidad del sistema de mensajería**	colas, consumidores, lag, rendimiento, capacidad, alertas	"Alertas en el lag (número de mensajes en cola) para anticipar problemas". "Una cola de mensajes con varios consumidores necesita métricas de rendimiento para dimensionar el sistema". "En Yuno medimos el tiempo que tarda cada evento en propagarse; esa latencia debería verse en la herramienta de pruebas". "En Antares probamos colas militares con mensajes en ráfaga, y sin métricas de saturación es imposible anticipar fallas".

El análisis de las entrevistas permitió identificar seis necesidades clave en los procesos de prueba de arquitecturas orientadas a eventos. Los participantes coincidieron en que los principales desafíos se relacionan con la falta de trazabilidad y observabilidad de extremo a extremo, la escasa automatización de las herramientas actuales y la dificultad para reproducir errores intermitentes que surgen únicamente bajo condiciones de carga real.
Asimismo, se destacó la debilidad en la gestión de ambientes y datos de prueba, lo que compromete la reproducibilidad, y la ausencia de una herramienta integral que combine validación de contratos, trazabilidad visual y métricas de desempeño.
Estos hallazgos (consistentes entre sectores y roles) reafirman lo planteado por la literatura sobre la necesidad urgente de enfoques especializados para pruebas en sistemas basados en eventos (Wu et al. (2022); Silva et al. (2025)).

3.2.2. Análisis de encuestas

3.2.2.1. Metodología de análisis

Con el fin de complementar el análisis cualitativo obtenido a partir de las entrevistas, los resultados de la encuesta fueron analizados mediante un enfoque mixto que integró métodos descriptivos, inferenciales y de minería de texto. Este enfoque permitió obtener una visión más amplia y cuantitativamente sustentada de los patrones observados, garantizando la triangulación de los datos recolectados.
Para las preguntas con escala Likert se aplicó un análisis descriptivo e inferencial no paramétrico, acorde con las recomendaciones de Alkharusi (2022). Las variables categóricas y de selección múltiple se procesaron mediante frecuencias, matrices de coocurrencias y pruebas de independencia χ^2, siguiendo los procedimientos de McHugh (2013); Chen and Liu (2020); Koziol and Bilder (2014). Finalmente, las respuestas abiertas fueron analizadas mediante técnicas de minería de texto basadas en TF-IDF y K-Means, conforme a la

metodología propuesta por Sampath and Vignesh (2024).
En la Tabla 4 se resumen las estrategias metodológicas aplicadas al análisis de los resultados de la encuesta, diferenciadas según el tipo de variable.

Tabla 4: Estrategias aplicadas al análisis de los resultados de la encuesta

Tipo de pregunta	**Estrategia de análisis**	**Propósito del análisis**
Categóricas nominales (P1, P2, P5)	**Prueba de independencia** χ^2 y análisis descriptivo (frecuencias y porcentajes).	Identificar asociaciones entre variables como el rol, la experiencia o la ocurrencia de fallos en producción, además de describir la distribución general de la muestra.
Ordinales tipo Likert (P3, P4)	**Análisis descriptivo e inferencial no paramétrico**: medidas de tendencia central (media, mediana) y correlación de Spearman.	Explorar patrones en la percepción de confiabilidad y frecuencia de errores, respetando la naturaleza ordinal de las escalas Likert.
Comparación entre grupos (experiencia, rol)	**Pruebas de Mann–Whitney U y Kruskal–Wallis.**	Comparar la distribución de respuestas Likert entre grupos (por ejemplo, con o sin experiencia en sistemas RabbitMQ).
Preguntas de selección múltiple (P6, P7)	**Análisis de frecuencias múltiples** y **matriz de co-ocurrencias** de respuestas binarias.	Identificar los problemas y herramientas más frecuentes, además de detectar combinaciones recurrentes de dificultades en sistemas basados en eventos.
Pregunta abierta (P8)	**Análisis de contenido temático asistido por IA** mediante vectorización **TF-IDF** y agrupamiento **K-Means.**	Agrupar las respuestas textuales según temas comunes (por ejemplo, trazabilidad, observabilidad o validación de contratos) y sintetizar percepciones cualitativas.

3.2.2.2. Análisis

Para facilitar el procesamiento y la interpretación sistemática de los datos recolectados en la encuesta, se implementó un proceso de análisis asistido por inteligencia artificial (IA) mediante un modelo de lenguaje (LLM) orientado por un conjunto de instrucciones específicas (prompt) detalladas en el Anexo D. Este enfoque permitió automatizar el cálculo de estadísticas descriptivas e inferenciales, así como la detección de patrones semánticos en las respuestas abiertas, garantizando consistencia y trazabilidad en el tratamiento de los datos.

El modelo aplicó las técnicas de análisis descritas en la subsección anterior (incluyendo pruebas estadísticas, análisis de correlación y agrupamiento temático) sobre las respuestas recolectadas, integrando los resultados de forma estructurada. De esta manera, el análisis de la encuesta complementa los hallazgos cualitativos obtenidos en las entrevistas.

A continuación, se presentan los resultados derivados de este proceso analítico por etapas de análisis:

Primera etapa: Análisis de variables categóricas

Se aplicaron las técnicas estadísticas descritas previamente para analizar las variables categóricas de la encuesta y explorar posibles asociaciones entre ellas.

Tabla 5: Distribución y asociaciones entre variables categóricas

Variable	Distribución	Resultados
Rol (P1)	Otro: 36 (46.15 %) Backend Developer: 17 (21.79 %) Backend/Frontend Lead: 9 (11.54 %) Product Manager: 7 (8.97 %) DevOps / SRE: 3 (3.85 %) Arquitecto de Software: 3 (3.85 %) QA Engineer: 2 (2.56 %) QA Lead: 1 (1.28 %)	–
Experiencia (P2)	No: 59 (75.64 %) Sí: 19 (24.36 %)	–
Fallos en producción (P5)	No estoy seguro: 61 (78.21 %) Sí: 15 (19.23 %) No: 2 (2.56 %)	–

Rol × Experiencia	Asociación entre tipo de rol y experiencia.	$\chi^2 = 14{,}88$ gl = 7 p = 0.0375 V = 0.44
Experiencia × Fallos	Asociación entre experiencia y fallos en producción.	$\chi^2 = 67{,}50$ gl = 2 p <0.001 V = 0.93

A continuación, se describen los estadísticos utilizados en la tabla:

- χ^2 **(Chi-cuadrado):** mide la diferencia entre las frecuencias observadas y las esperadas bajo la hipótesis de independencia entre variables categóricas.
- **gl (grados de libertad):** número de comparaciones independientes posibles dentro de la tabla de contingencia.
- **p (nivel de significancia):** probabilidad de que el resultado observado ocurra por azar; valores menores a 0.05 indican asociación significativa.
- **V (Cramér):** medida del tamaño del efecto o fuerza de asociación entre variables; varía entre 0 (sin asociación) y 1 (asociación perfecta).

La Tabla 5 presenta la distribución de las variables categóricas y las asociaciones entre ellas. El rol predominante corresponde a Otro (46.15 %), seguido de Backend Developer (21.79 %). Solo el 24.36 % de los encuestados indicó tener experiencia en pruebas de sistemas basados en eventos o RabbitMQ.
Respecto a los fallos en producción, el 78.21 % manifestó no estar seguro de haber experimentado incidentes relacionados con fallas asíncronas no detectadas durante las pruebas.
El análisis de independencia mediante la prueba de χ^2 evidenció una asociación moderada entre el tipo de rol y la experiencia ($V = 0{,}44$, $p = 0{,}0375$). Asimismo, se observó una asociación muy fuerte entre la experiencia y la ocurrencia de fallos en producción ($V = 0{,}93$, $p < 0{,}001$), según la clasificación de Cramér.
Estos resultados sugieren que los participantes con mayor experiencia en arquitecturas orientadas a eventos poseen una capacidad superior para identificar fallos asíncronos no cubiertos por las pruebas tradicionales, lo que refuerza la necesidad de herramientas de validación más especializadas para este tipo de sistemas.

Segunda etapa: Análisis de variables ordinales (escala Likert)

Luego de analizar las variables categóricas, esta etapa amplía el análisis hacia las variables ordinales de la encuesta, con el propósito de explorar la percepción de confiabilidad y la frecuencia de aparición de problemas difíciles de depurar en los entornos evaluados.

Tabla 6: Estadísticos descriptivos y correlación entre variables tipo Likert

Variable	**Media**	**Mediana**	**Moda**	**DE**	**IQR**	**N válidos**
P3 – Frecuencia de problemas difíciles de depurar	3.21	3	3	0.42	0.00	19
P4 – Confiabilidad de las herramientas para detectar fallas asincrónicas	4.11	4	4	0,57	0.00	19
Correlación de Spearman: $\rho = -0{,}12$						

En este análisis se obtuvo que la frecuencia para problemas difíciles de depurar (P3), la media fue de 3.21, correspondiente a la categoría 'ocasionalmente', con baja dispersión (DE = 0.42), lo que indica que la mayoría de los participantes experimenta este tipo de problemas de forma moderada y homogénea.

Por su parte, la confiabilidad percibida de las herramientas (P4) presentó una media de 4.11 y una mediana de 4, reflejando una valoración predominantemente positiva hacia las herramientas utilizadas. La baja variabilidad (DE = 0.57, IQR = 0) sugiere un consenso general respecto a su confiabilidad.

La correlación de Spearman ($\rho = -0{,}12$) mostró una relación negativa muy débil y no significativa entre ambas variables. Esto indica que la frecuencia con que se presentan problemas difíciles de depurar no se asocia de manera sustancial con la percepción de confiabilidad de las herramientas.

Este resultado sugiere que los encuestados tienden a mantener una opinión estable sobre la confiabilidad de sus herramientas, independientemente de la frecuencia con que enfrentan fallas. Esto podría indicar que la percepción de confiabilidad depende más de la familiaridad con las herramientas que de su desempeño real en escenarios de prueba, reforzando la necesidad de soluciones que ofrezcan métricas objetivas de validación en sistemas basados en eventos.

Tercera etapa: Análisis de preguntas de selección múltiple

Esta etapa complementa los análisis previos de variables categóricas y ordinales, abordando las preguntas de selección múltiple con el propósito de identificar las principales dificultades enfrentadas al probar sistemas basados en eventos y las herramientas más utilizadas para dichas pruebas. Este análisis permitió observar combinaciones frecuentes entre problemas y soluciones.

Tabla 7: Análisis de preguntas de selección múltiple: dificultades y herramientas utilizadas

Opción	**% de aparición**	**Principales co-ocurrencias**	**Interpretación**
P6 — Dificultades enfrentadas al probar sistemas basados en eventos			
Eventos perdidos	16.67 %	Mensajes duplicados (9), Errores intermitentes (6)	Principal dificultad reportada; evidencia fallas en el aseguramiento de entrega de mensajes en arquitecturas asíncronas.
Mensajes duplicados	15.38 %	Eventos perdidos (9), Condiciones de carrera (5)	Problema recurrente que compromete la consistencia del sistema y las pruebas de idempotencia.
Condiciones de carrera	10.26 %	Errores intermitentes (5), Eventos perdidos (5)	Asociada a la concurrencia y asincronía; difícil de reproducir en entornos de prueba.
Errores intermitentes	8.97 %	Eventos perdidos (6), Condiciones de carrera (5)	Indica inestabilidad o comportamiento no determinista durante la ejecución de eventos.

Falta de trazabilidad	7.69 %	Eventos perdidos (6), Mensajes duplicados (4)	Muestra la necesidad de observabilidad y seguimiento de eventos entre componentes.
Otros	1.28 %	–	Respuestas atípicas o de baja frecuencia.
Combinaciones más frecuentes (P6)			
1) Eventos perdidos + Mensajes duplicados (9) 2) Errores intermitentes + Condiciones de carrera (6) 3) Eventos perdidos + Falta de trazabilidad (6)			
P7 — Herramientas utilizadas en pruebas de sistemas basados en eventos			
Postman	19.23 %	Jest/Mocha/Chai (5), RabbitMQ Management Plugin (5)	Herramienta genérica más usada, indica ausencia de soluciones especializadas en testing de eventos.
RabbitMQ Management Plugin	10.26 %	Postman (5), Jest/Mocha/Chai (3)	Uso frecuente para inspección manual de colas y mensajes.
Jest/Mocha/Chai	6.41 %	Postman (5), RabbitMQ Management Plugin (3)	Pruebas unitarias o de integración limitadas a componentes individuales.
Otras	6.41 %	Postman (4), Wireshark (1)	Incluye herramientas no estandarizadas, sugiere exploración de soluciones ad-hoc.

Wireshark	3.85 %	Postman (2), Otras (1)	Usada para inspección de tráfico, más cercana al nivel de red.
Pact / Contract Testing	1.28 %	Postman (1)	Uso marginal; muestra escasa adopción de pruebas basadas en contratos en EDA.
Combinaciones más frecuentes (P7)			
1) Jest/Mocha/Chai + Postman (5) 2) Postman + RabbitMQ Management Plugin (5) 3) Otras + Postman (4)			

En las dificultades (P6), los problemas más frecuentes fueron eventos perdidos (16.67 %), mensajes duplicados (15.38 %) y condiciones de carrera (10.26 %). Estos resultados evidencian las limitaciones de las estrategias de prueba tradicionales para abordar aspectos críticos de las arquitecturas orientadas a eventos, como la asincronía, la concurrencia y la observabilidad.

En cuanto a las herramientas utilizadas (P7), Postman fue la más reportada (19.23 %), acompañada de utilidades del broker (RabbitMQ Management Plugin) y marcos de pruebas unitarias (Jest/Mocha/Chai). El bajo uso de herramientas especializadas, como Pact, refleja una brecha significativa en la adopción de enfoques basados en contratos y en la simulación integral de flujos de eventos.

Las combinaciones más frecuentes entre herramientas y dificultades confirman la necesidad de una solución integrada que permita realizar validaciones confiables de entrega, verificación de idempotencia y trazabilidad centralizada de los mensajes dentro de entornos asincrónicos.

Asimismo, los resultados de esta etapa guardan coherencia con los hallazgos cualitativos de las entrevistas, en los que los participantes destacaron las mismas limitaciones en trazabilidad, observabilidad y automatización. Esta correspondencia refuerza la validez de los resultados y la pertinencia de avanzar hacia el diseño de una herramienta de pruebas especializada para arquitecturas orientadas a eventos.

Cuarta etapa: Análisis de preguntas abiertas

Finalmente, esta etapa complementa los análisis previos abordando la pregunta abierta (P8) de la encuesta. Su propósito fue identificar, mediante técnicas de minería de texto,

los temas recurrentes en las percepciones de los participantes sobre los retos y expectativas asociados a las pruebas en arquitecturas orientadas a eventos.

Tabla 8: Agrupación temática de respuestas abiertas mediante análisis TF–IDF y K–Means

Clúster	**Tema principal**	**Palabras clave (mayor peso TF–IDF)**	**Citas representativas**
1	Trazabilidad y observabilidad	comentarios, trazabilidades, documentación, logs centralizados, facilidad	"Logs centralizados." "Buena observabilidad en sus componentes."
2	Facilidad de integración	fácil, integración, fácil integración, microservicios, orientadas, eventos	"Una fácil integración con microservicios que están orientadas a eventos." "Fácil integración."
3	Recuperación y consistencia de eventos	eventos, pérdida, problemas, recuperación, conexión momentánea, transmisión	"Recuperación de eventos perdidos en caso de problemas de transmisión o pérdida momentánea de conexión." "Poder emular la recepción de eventos de forma concurrente."
4	Trazabilidad y monitoreo con herramientas externas	trazabilidad, integrar datadog, visibilidad, ranbitmq, monitoreo, observabilidad	"Que la trazabilidad sea sencilla y esté configurada por defecto. En nuestro caso integramos Datadog con RabbitMQ para tener mayor visibilidad." "Trazabilidad."

La Tabla 8 presenta los resultados del análisis de las respuestas abiertas (P8). En conjunto, los hallazgos reflejan tres ejes conceptuales principales:

1. Trazabilidad y observabilidad: los participantes destacaron la necesidad de centralizar logs y mejorar la visibilidad del flujo de eventos entre componentes.

2. Facilidad de integración: se valoró la capacidad de integrar herramientas de prueba con ecosistemas de microservicios ya existentes.

3. Consistencia y recuperación de eventos: se enfatizó la importancia de asegurar la entrega y recuperación confiable de mensajes en escenarios de pérdida o desconexión momentánea.

Estos resultados cualitativos complementan los análisis cuantitativos previos, al profundizar en las percepciones y necesidades expresadas directamente por los participantes. Los temas identificados coinciden con las dificultades observadas en la encuesta (como la falta de trazabilidad y la detección de errores intermitentes) y con los hallazgos de las entrevistas, consolidando la evidencia sobre la necesidad de herramientas integradas que combinen trazabilidad, monitoreo y validación automatizada de eventos en arquitecturas basadas en eventos.

3.2.2.3. Observación directa / experiencias previas

Además del análisis de entrevistas, encuestas y revisión documental, se consideró la observación directa basada en experiencias profesionales del investigador, con el propósito de contrastar los hallazgos obtenidos con situaciones reales de desarrollo y pruebas en entornos empresariales.
En el ámbito profesional, el investigador ha participado en proyectos de empresas de distintos tamaños y niveles de madurez tecnológica, como Ecobot, Rappi, Mercado Libre y Yuno. En dichos contextos se trabajó con arquitecturas EDA mediante intermediarios de mensajería ampliamente usados en la industria, entre ellos Kafka, RabbitMQ, Amazon MQ y Google Cloud Pub/Sub, estos dos últimos a través de SDKs desarrollados internamente por las compañías.

En el caso de Ecobot, una startup con procesos de desarrollo en consolidación, se evidenció la ausencia de herramientas robustas para la validación de flujos punto a punto en arquitecturas EDA. Sin un ecosistema de DevOps ni frameworks especializados en pruebas, las validaciones se realizaban de manera manual. El procedimiento consistía en aislar el microservicio en desarrollo, desplegarlo en un entorno local y, mediante el administrador del intermediario de mensajería, validar el comportamiento de los mensajes (por ejemplo, la llegada o pérdida de eventos). Posteriormente, se observaba la reacción de los demás microservicios del flujo ante los cambios introducidos. Este proceso reveló un aspecto crítico: una modificación puntual podía requerir ajustes adicionales en varios servicios, incrementando la complejidad de las pruebas y el mantenimiento.

En Mercado Libre, un entorno corporativo con procesos maduros y herramientas sofisticadas, la problemática persistía. A pesar de contar con flujos de integración continua, entornos distribuidos y prácticas avanzadas de monitoreo, las pruebas sobre flujos asincrónicos mantenían un carácter sectorizado: primero se validaba el servicio modificado

y luego, progresivamente, su integración con los demás microservicios del flujo.

A partir de las experiencias relatadas, se identificaron patrones comunes en organizaciones de distinta escala:

- Carencia de herramientas de pruebas especializadas para arquitecturas EDA.
- Necesidad de validación progresiva de microservicios en flujos asincrónicos.
- Riesgo de fallos no deterministas por condiciones de carrera o errores intermitentes.

Estos hallazgos coinciden con la literatura, que enfatiza la falta de metodologías estandarizadas y entornos automatizados capaces de abordar la complejidad de la asincronía en microservicios (Silva et al. (2025); Wu et al. (2022); Herbold and Harms (2013)).
En conjunto, estas observaciones complementan los resultados empíricos y fortalecen la definición del problema central, que servirá como punto de partida para la formulación de los requerimientos funcionales de la herramienta propuesta.

3.2.3. Análisis de documentación

Como parte del proceso de definición del problema, se analizó la documentación técnica de herramientas de pruebas ampliamente utilizadas en la industria, con el propósito de contrastar sus capacidades frente a los requerimientos identificados para las arquitecturas orientadas a eventos. Este análisis complementa los resultados de las encuestas y entrevistas, al verificar si las soluciones más adoptadas ofrecen mecanismos adecuados para validar flujos asincrónicos y garantizar la trazabilidad de eventos.
En el caso de Postman, la documentación pública incluye colecciones como RabbitMQ API, que facilitan la interacción con la API HTTP de RabbitMQ. Sin embargo, estas funcionalidades se limitan a solicitudes REST y no proveen mecanismos específicos para probar flujos asincrónicos de mensajería AMQP (Postman (2025)).
En cuanto a JMeter, aunque se trata de una herramienta de pruebas de rendimiento ampliamente versátil según su manual oficial (Apache Software Foundation (2025)) el soporte para RabbitMQ depende exclusivamente de extensiones. Por ejemplo, el JMeter-Rabbit-AMQP Plugin permite crear publicadores y consumidores, pero requiere instalación manual en el directorio de librerías de JMeter. Este requisito evidencia que el soporte para mensajería asíncrona no está integrado de forma nativa, lo que incrementa la complejidad y limita su adopción en entornos de integración continua.
La siguiente tabla sintetiza los hallazgos más relevantes de este análisis:

Tabla 9: Comparación de Postman y JMeter frente al soporte de flujos asincrónicos con RabbitMQ

Herramienta	Capacidades principales	Limitaciones en entornos EDA
Postman	Documentación extensa para pruebas de APIs REST. Colecciones preconfiguradas para interactuar con servicios HTTP, incluyendo la API de RabbitMQ.	No ofrece soporte nativo para protocolos de mensajería asincrónica como AMQP. Se restringe a escenarios síncronos mediante peticiones HTTP.
JMeter	Herramienta de referencia en pruebas de rendimiento, extensible mediante plugins.	El soporte para RabbitMQ depende de extensiones externas. Requiere instalación manual de librerías adicionales, lo cual incrementa la complejidad y limita su integración fluida en pipelines DevOps.

En conclusión, aunque Postman y JMeter son herramientas consolidadas y ampliamente adoptadas, su cobertura en el contexto de arquitecturas orientadas a eventos resulta limitada. Ninguna de ellas ofrece soporte nativo para la validación de flujos asincrónicos, trazabilidad de eventos punto a punto ni verificación de contratos de mensajes. Este vacío funcional, identificado también en las entrevistas y la literatura revisada, refuerza la necesidad de una herramienta especializada que integre monitoreo, validación y trazabilidad de eventos en entornos EDA.

3.2.4. Síntesis de hallazgos

La agrupación de los resultados obtenidos en esta fase permitió consolidar una visión precisa de la problemática que enfrentan los equipos de desarrollo y aseguramiento de calidad al trabajar con arquitecturas orientadas a eventos (EDA).
Los distintos instrumentos de recolección de información (entrevistas, encuestas, análisis documental y observación directa) coincidieron en señalar deficiencias estructurales en los procesos de prueba de sistemas asincrónicos, entre ellas:

- Ausencia de herramientas especializadas que soporten validación de flujos de eventos, trazabilidad punto a punto y verificación de contratos de mensajes.
- Predominio de procesos de validación manual o parcialmente automatizada, con cobertura limitada frente a escenarios de concurrencia o fallos intermitentes.
- Dificultad para reproducir errores asincrónicos en ambientes de prueba y falta de mecanismos de observabilidad integrados.

- Dependencia de soluciones genéricas (Postman, JMeter) o plugins aislados sin soporte nativo para mensajería AMQP.

3.2.5. Formulación del problema

A partir de los hallazgos, se definió el problema central que orienta la siguiente fase del proceso metodológico:

"Las herramientas de pruebas actualmente utilizadas en entornos de desarrollo no permiten validar, monitorear ni depurar de forma automatizada los flujos de eventos en arquitecturas orientadas a eventos, lo que dificulta la trazabilidad, la detección temprana de errores y la confiabilidad del sistema."

En consecuencia, se establecieron los requerimientos funcionales generales que deberá abordar la propuesta de solución:

1. Incorporar mecanismos de trazabilidad centralizada y observabilidad de eventos entre microservicios.
2. Permitir la validación automatizada de contratos de mensajes y la detección de incompatibilidades de payloads o routing keys.
3. Ofrecer capacidades de simulación y monitoreo en tiempo real para la evaluación del rendimiento y confiabilidad del flujo de eventos.
4. Integrarse fácilmente con entornos de desarrollo y flujos DevOps existentes.

Esta definición constituye el resultado final de la fase de Definir y servirá como punto de partida para la fase de Idear, en la cual se generarán y evaluarán alternativas de diseño para una herramienta que dé respuesta a las necesidades aquí identificadas.

3.2.6. Necesidades priorizadas

A partir de los resultados obtenidos en la fase de Definir, se consolidó y priorizó un conjunto de necesidades que orientó el proceso de ideación de la solución. La priorización se realizó considerando tres criterios principales: el impacto esperado en la calidad del proceso de pruebas, la frecuencia con que las necesidades fueron mencionadas por los participantes y su alineación directa con los objetivos del proyecto.
Las necesidades priorizadas fueron las siguientes:

- **Trazabilidad de eventos:** Identificada como la necesidad más crítica, debido a que la ausencia de mecanismos para rastrear el recorrido completo de un evento entre servicios dificulta el diagnóstico de fallos intermitentes y la validación de entregas.

- **Validación de contratos entre productores y consumidores:** Se estableció la importancia de contar con verificaciones estructurales y semánticas de los mensajes intercambiados, con el fin de reducir errores silenciosos y mejorar la interoperabilidad entre servicios.
- **Simulación de escenarios reales de mensajería:** Se reconoció la necesidad de replicar condiciones de operación con múltiples productores y consumidores, para evaluar el comportamiento del sistema en contextos representativos de carga y concurrencia.
- **Monitoreo de métricas de calidad:** Se priorizó la capacidad de medir variables como tiempo de entrega, duplicidad de mensajes y frecuencia de errores, con el propósito de caracterizar la confiabilidad y el rendimiento de los flujos asincrónicos.
- **Integración con el ecosistema de desarrollo:** Se observó la necesidad de disponer de una herramienta que pudiera incorporarse fácilmente a los entornos de los equipos de desarrollo, sin configuraciones complejas ni dependencias externas.
- **Estandarización y documentación del proceso de prueba:** Se destacó la importancia de establecer lineamientos claros para la verificación de flujos de eventos y el registro de resultados, fortaleciendo la mantenibilidad y trazabilidad del conocimiento técnico.

El resultado de esta priorización permitió definir el enfoque de la solución: una herramienta modular que abordara de forma directa las cuatro primeras necesidades, mientras que las dos restantes se integraron como atributos transversales de calidad dentro del diseño arquitectónico. Esta priorización constituyó el punto de partida para la generación de ideas y alternativas de diseño, desarrolladas en los apartados siguientes.

3.3. Fase de Idear

El propósito de esta fase fue generar y evaluar alternativas técnicas que respondieran de manera directa a las necesidades identificadas en las fases previas de Empatizar y Definir, considerando su viabilidad técnica, la calidad del producto esperado y su alineación con los objetivos del proyecto.
A partir de los hallazgos consolidados en la fase anterior, se formularon diversas propuestas de diseño orientadas a resolver las deficiencias detectadas en la validación y trazabilidad de flujos asincrónicos. Estas alternativas fueron analizadas de acuerdo con criterios de impacto esperado, factibilidad de implementación y grado de alineación con los requerimientos funcionales definidos previamente.
Como resultado de este proceso, se seleccionó una arquitectura modular como base de la herramienta propuesta. Este enfoque permitió definir una solución escalable, de fácil mantenimiento y adaptable a distintos entornos organizacionales, además de garantizar su integración con RabbitMQ.
En los apartados siguientes se describen las necesidades priorizadas, la conceptualización de la propuesta y la justificación de las decisiones arquitectónicas que orientaron el diseño de la herramienta. Esta fase fue las bases para la etapa de Prototipar, donde se materializó la solución conceptual en un prototipo funcional validado en entornos controlados.

3.3.1. Generación de ideas

Con base en las necesidades priorizadas, se llevó a cabo una lluvia de ideas orientada a generar posibles alternativas de solución y funcionalidades para la herramienta desarrollada. El propósito fue fomentar una generación libre y divergente de propuestas, sin restricciones iniciales de viabilidad técnica, promoviendo la exploración de distintas perspectivas sobre el problema. Entre las propuestas se destacaron las siguientes:

1. Conectar y administrar diferentes intermediarios de mensajería.
2. Obtener métricas de los intermediarios de mensajería.
3. Guardar flujos de pruebas como sesiones de trabajo.
4. Simular el flujo completo de eventos entre servicios, observando su comportamiento en tiempo real.
5. Incorporar un componente de validación de contratos basado en especificaciones de AsyncAPI.
6. Permitir validaciones manuales como alternativa a AsyncAPI.

7. Implementar un sistema de monitoreo de métricas capaz de registrar tiempos de entrega, duplicidad de mensajes y frecuencia de errores.
8. Incluir un tablero de reportes visuales para la interpretación de resultados y trazabilidad de flujos de prueba.
9. Disponer de un panel de inicio de sesión.
10. Integrarse con sistemas CI/CD para la ejecución automatizada de flujos de prueba.
11. Enviar notificaciones en tiempo real ante fallos o inconsistencias detectadas.
12. Visualizar gráficamente los flujos de eventos entre productores y consumidores.

Para seleccionar las ideas más viables, se aplicó una matriz de priorización, en la que cada propuesta fue evaluada según tres criterios: impacto esperado en la calidad del proceso de pruebas, viabilidad técnica de implementación y alineación con los objetivos del proyecto.

Tabla 10: Matriz de priorización de ideas

Idea propuesta	**Impacto**	**Viabilidad técnica**	**Alineación con el proyecto**	**Puntaje total**
Conectar y administrar diferentes intermediarios de mensajería	4	5	5	14
Obtener métricas de los intermediarios de mensajería	4	4	4	12
Guardar flujos de pruebas como sesiones de trabajo	4	5	4	13
Simular el flujo completo de eventos entre servicios	5	4	5	14
Validar contratos basados en especificaciones AsyncAPI	5	4	5	14
Generar validaciones manuales como alternativa a AsyncAPI	3	5	4	12
Monitorear métricas de desempeño y errores en los flujos	5	4	5	14

Tablero de reportes visuales para interpretación y trazabilidad	4	4	5	13
Panel de inicio de sesión	4	5	4	13
Integración con sistemas de CI/CD para pruebas automatizadas	5	2	5	12
Notificaciones en tiempo real ante fallos o inconsistencias	4	4	4	12
Visualización gráfica de flujos de eventos	5	3	4	12

Al analizar los resultados de la matriz, se estableció un umbral de priorización de 13 puntos o más (equivalente al 85 % del puntaje máximo posible). Este rango correspondió a las ideas con alta prioridad. Estas ideas sirvieron de base para la conceptualización de la solución técnica desarrollada a continuación.

3.3.2. Conceptualización de la propuesta de solución

La propuesta se diseñó con la capacidad de integrarse con RabbitMQ y adaptarse a distintos entornos de desarrollo. Cada módulo fue definido para cumplir una función específica dentro del proceso de pruebas, favoreciendo su mantenibilidad, reutilización y evolución independiente. El concepto central de la herramienta combina dos enfoques complementarios de validación:

- **Validación de contratos:** destinada a verificar la coherencia estructural y semántica de los mensajes entre productores y consumidores.
- **Modelado de estados y transiciones:** enfocado en representar y analizar el flujo de eventos a lo largo del sistema, permitiendo identificar desviaciones o comportamientos no esperados.

De esta manera, la herramienta busca ofrecer una visión punto a punto del proceso de pruebas, abarcando desde la generación de eventos hasta el análisis de resultados.
Durante la conceptualización, se definieron los siguientes principios orientadores del diseño:

1. **Independencia tecnológica:** garantizar la interoperabilidad con distintos servicios sin depender de lenguajes o frameworks específicos.

2. **Automatización de pruebas:** reducir la intervención manual y asegurar consistencia en la ejecución.

3. **Observabilidad y transparencia:** permitir el seguimiento detallado del ciclo de vida de cada evento.

4. **Facilidad de adopción:** asegurar una integración fluida en entornos ágiles sin cambios significativos en la infraestructura.

En síntesis, la propuesta se concibió como una plataforma integral de pruebas para arquitecturas orientadas a eventos, capaz de validar contratos, simular escenarios reales y generar métricas cuantitativas sobre el desempeño y la confiabilidad del sistema. Esta conceptualización se constituyó en la base para la siguiente fase metodológica, Prototipar, donde se materializó el diseño en un prototipo funcional.

3.3.3. Criterios de evaluación de alternativas de diseño

Una vez conceptualizada la propuesta de solución, fue necesario evaluar las posibles arquitecturas y tecnologías que podrían soportarla, con el fin de seleccionar la alternativa más adecuada para su desarrollo. Para ello, se establecieron una serie de criterios de evaluación que permitieron comparar de manera objetiva las distintas opciones consideradas.
Estos criterios se definieron a partir de las necesidades priorizadas anteriormente, complementadas con los principios de calidad del software propuestos por el Consortium for Information & Software Quality (CISQ) (2022) y las buenas prácticas de ingeniería de software descritas por Vernon en Domain-Driven Design Distilled (2016).

Los criterios definidos fueron los siguientes:

1. **Facilidad de integración:** Se valoró la capacidad de cada alternativa para integrarse con los entornos de desarrollo y mensajería existentes, especialmente con los administradores de mensajería como RabbitMQ.

2. **Escalabilidad:** Se analizó el potencial de la arquitectura para soportar un incremento en el número de servicios, eventos y consumidores sin degradar el rendimiento del sistema de pruebas.

3. **Mantenibilidad:** Se consideró la facilidad para extender, modificar o depurar los componentes de la herramienta sin afectar su funcionamiento global.

4. **Automatización:** Se evaluó el grado de automatización que cada alternativa ofrecía en la generación, ejecución y validación de pruebas asincrónicas.

5. **Observabilidad:** Se valoró la capacidad para monitorear métricas clave de desempeño (tiempos de entrega, duplicidad, pérdidas de eventos) y generar reportes que facilitaran el diagnóstico de fallos.

6. **Compatibilidad tecnológica:** Se analizó la interoperabilidad con diferentes servicios, manteniéndose agnóstico a los lenguajes de programación o frameworks utilizados, con el propósito de garantizar la adopción en diversos contextos.

7. **Viabilidad técnica:** Se consideró la disponibilidad de recursos, bibliotecas y documentación que facilitaran la implementación de cada alternativa dentro del alcance temporal y técnico del proyecto.

La aplicación de estos criterios permitió realizar una evaluación equilibrada entre los aspectos funcionales, técnicos y de calidad, garantizando que la alternativa seleccionada respondiera tanto a las necesidades detectadas como a los objetivos académicos y prácticos del proyecto. De esta manera, se aseguraron decisiones fundamentadas para la selección final del diseño arquitectónico y tecnológico que sustenta la herramienta.

3.3.4. Alternativas analizadas

Con el propósito de seleccionar la arquitectura más adecuada para la herramienta, se analizaron distintas opciones de diseño aplicando los criterios definidos previamente (integración, escalabilidad, mantenibilidad, automatización, observabilidad, compatibilidad tecnológica y viabilidad técnica). El objetivo fue balancear la rapidez de implementación y la facilidad de adopción con la calidad técnica esperada.

1. **Monolito con diseño modular interno (seleccionada).** Esta alternativa consistió en una única aplicación desplegable que concentró todas las capacidades (validación de contratos, simulación de eventos, trazabilidad y métricas), organizada en módulos internos bien definidos (capas y componentes independientes a nivel de código). Su principal ventaja fue la velocidad de implementación y la simplicidad operativa. Además, facilitó la adopción por equipos que no requerían operar múltiples servicios adicionales. Como desventaja potencial, el escalamiento horizontal sería menos flexible que en un esquema distribuido; no obstante, resultó suficiente para el alcance del proyecto y los escenarios de prueba planteados.

2. **Arquitectura modular orquestada por eventos entre módulos (distribuida).** Esta alternativa planteaba módulos desacoplados que se comunicaron entre sí a través de un intermediario de mensajería (p. ej., RabbitMQ), utilizando eventos para orquestar flujos internos de la herramienta. El enfoque mantenía coherencia con

principios EDA y facilitaba extensibilidad y reemplazo de componentes. Sin embargo, no era estrictamente necesario que la herramienta de pruebas fuese distribuida, ya que su misión era probar sistemas distribuidos, no replicar esa distribución en su propio diseño. Supune mayor complejidad de operación y despliegue (más servicios que instalar, configurar y monitorear).

3. **Microservicios acoplados vía interfaces HTTP.** Cada capacidad (validación, simulación, monitoreo) se concebía como servicio independiente comunicándose por HTTP. Aunque mejora la separación de responsabilidades frente al monolito, introduce más puntos de fallo, mayores costos de integración y coordinación, y una complejidad operativa que no aportaba valor proporcional al objetivo del proyecto (tiempos y alcance).

A partir de los criterios definidos, se compararon las alternativas como se resume en la Tabla 11.

Tabla 11: Evaluación comparativa de alternativas arquitectónicas (revisada)

Criterio	**Monolito modular**	**Modular orquestada por eventos**	**Microservicios HTTP**
Facilidad de integración	5	4	4
Escalabilidad	4	5	4
Mantenibilidad	5	5	4
Automatización	5	5	3
Observabilidad	5	5	4
Compatibilidad tecnológica	5	5	4
Viabilidad técnica (tiempo/recursos)	4	3	4
Puntaje total	**33**	**32**	**27**

Como se observa en la Tabla 11, el monolito con diseño modular interno obtuvo la mejor valoración global, impulsado por su rapidez de implementación, simplicidad de despliegue y alineación con el alcance del proyecto. Si bien la alternativa distribuida orquestada por eventos ofrecía un techo de escalabilidad superior, introducía complejidad operativa innecesaria para una herramienta cuyo propósito fue probar sistemas distribuidos y no reproducir su arquitectura. Por estas razones, se seleccionó el monolito modular como base para la fase de Prototipar, dejando abierta la posibilidad de evolucionar hacia componentes desacoplados en trabajos futuros si los escenarios de carga y adopción así lo exigían.

3.4. Fase de Prototipar

La fase de prototipar tuvo como propósito implementar el diseño conceptual definido en la etapa de idear, mediante el desarrollo de un prototipo funcional de la herramienta de pruebas automatizadas. Esta fase permitió validar la viabilidad técnica de la solución propuesta, así como comprobar su capacidad para integrarse con entornos de desarrollo reales y cumplir con los objetivos de trazabilidad, validación y monitoreo identificados en las fases anteriores.
Durante esta etapa se implementó una versión inicial de la herramienta bajo un enfoque monolítico modular, estructurada en componentes independientes que facilitaron la organización del código, la mantenibilidad y la evolución futura del sistema. Cada módulo se diseñó para cumplir una función específica dentro del proceso de pruebas, permitiendo abordar de forma separada aspectos como la conexión con el intermediario de mensajería, la simulación de flujos de eventos, la validación de contratos y el monitoreo de métricas.
El proceso de desarrollo se llevó a cabo siguiendo principios de ingeniería de software y buenas prácticas de desarrollo, priorizando la simplicidad y la escalabilidad.

3.4.1. Diseño general del prototipo

El diseño del prototipo se fundamentó en los principios de *Domain-Driven Design* (DDD) propuestos por Vernon (2016), orientados a mantener una relación directa entre el modelo del dominio y su implementación técnica. Bajo este enfoque, la herramienta se estructuró a partir de contextos delimitados (*bounded contexts*) y un lenguaje ubicuo (*ubiquitous language*) que garantizó coherencia conceptual entre el código, la API (Application Programming Interface) y la interfaz de usuario.
El sistema se implementó con dos componentes principales:

1. Un *backend* monolítico modular, que concentró la lógica de negocio, la integración con el intermediario de mensajería y la generación de reportes.

2. Un *frontend* independiente, encargado de la interacción con el usuario final mediante la visualización de flujos, métricas y resultados de las pruebas.

3.4.1.1. Arquitectura general del *backend*

El *backend* se implementó como un monolito modular (ver Anexo E), donde cada módulo representa un contexto funcional independiente dentro del dominio. Este diseño permitió mantener bajo acoplamiento y alta cohesión entre componentes, favoreciendo la mantenibilidad y la extensibilidad del sistema.
La arquitectura se organizó en tres grupos principales de módulos:

- **Servicios de dominio:** agrupa los módulos centrales del dominio que encapsulan las reglas de negocio y la lógica de la aplicación:
  - **Autenticador:** gestiona la autenticación y validación de usuarios mediante JWT (Json Web Token).
  - **Administradores de mensajería:** administra la conexión con RabbitMQ, controlando la creación de colas, el consumo y la publicación de mensajes.
  - **Flujos:** orquesta la definición, ejecución y seguimiento de flujos de prueba asincrónicos, simulando interacciones entre productores y consumidores.
  - **Reportes:** gestiona la persistencia y recuperación de resultados de las pruebas, generando métricas consolidadas por flujo.
  - **Lector de documentación:** interpreta documentos AsyncAPI o especificaciones YAML para generar automáticamente descripciones de flujos y contratos.
  - **Base de datos:** módulo de persistencia principal, implementado con MongoDB como base de datos documental.
- **Módulos de entrada de la Plataforma:** integra los mecanismos de comunicación e interacción con el exterior:
  - **Servidor HTTP:** expone los endpoints REST del sistema, recibe las peticiones del *frontend* y canaliza las operaciones hacia los módulos de dominio.
- **Módulos de salida de la Plataforma:** agrupa los componentes encargados de la integración con servicios externos, la ejecución de flujos y la observabilidad:
  - **Ejecuciones:** coordina la ejecución de los flujos configurados, gestionando el envío y recepción de mensajes a través del intermediario RabbitMQ.
  - **Conector de Intermediario:** implementa la conexión directa con el intermediario de mensajería, manejando los canales, colas y confirmaciones de entrega.
  - **Conector R2:** gestiona la integración con el servicio *Cloudflare R2*, permitiendo el almacenamiento remoto de documentos Asyncapi.
  - **Servidor websocket:** permite la comunicación en tiempo real entre el *frontend* y el *backend*, actualizando el estado de las ejecuciones y los reportes en curso.
  - ***Logger:*** centraliza la trazabilidad del sistema, integrándose con *Grafana Loki* para registrar eventos, métricas de desempeño y errores de ejecución.

La Figura 3.1 presenta la arquitectura general de módulos del *backend*, evidenciando la interacción entre las capas de dominio, plataforma y servicios externos.

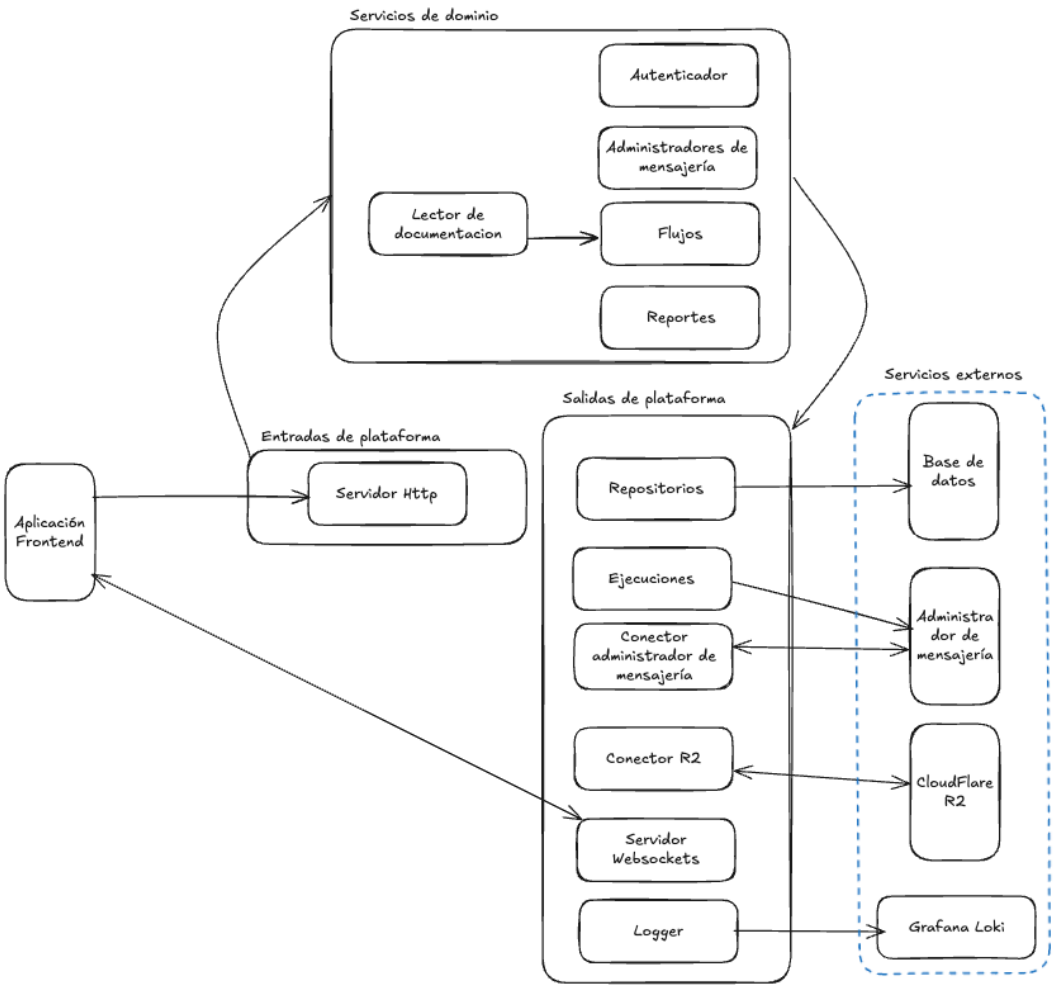


Figura 3.1: Arquitectura general de módulos del *backend*

Esta división modular refleja la aplicación del principio de contextos delimitados de Vernon (2016), donde cada módulo opera dentro de su propio espacio conceptual con modelos, contratos e interfaces independientes.
El lenguaje ubicuo, otro principio central del DDD, se implementó a través del uso consistente de términos compartidos entre el código, la API y la interfaz de usuario.

3.4.1.2. *Frontend* independiente

El *frontend* se implementó como un servicio desacoplado (ver Anexo F), encargado de consumir la API REST expuesta por el *backend*. Su propósito es ofrecer una interfaz vi-

sual para la gestión de pruebas, la administración de intermediarios de mensajería y la interpretación de resultados. La aplicación se estructuró en módulos por funcionalidades de la herramienta, manteniendo coherencia con los contextos del dominio definidos en el *backend*. Los módulos implementados fueron los siguientes:

- **Inicio de sesión:** gestiona la autenticación de usuarios mediante tokens JWT, validando credenciales y controlando el acceso a las funcionalidades protegidas.
- **Tablero principal:** actúa como el punto central de acceso, desde donde el usuario puede navegar hacia los diferentes módulos de la herramienta.
- **Intermediarios de mensajería:** permite registrar, visualizar y monitorear los intermediarios de mensajería disponibles, integrándose con los módulos de configuración del *backend*.
- **Flujos:** gestiona la definición, ejecución y seguimiento de flujos asincrónicos, coordinando las pruebas entre productores y consumidores. Desde este módulo se accede a las funcionalidades de Documentación y Ejecuciones.
- **Documentación:** permite cargar y visualizar especificaciones AsyncAPI, generando representaciones de los contratos de eventos y facilitando su validación durante las pruebas.
- **Ejecuciones:** muestra el estado en tiempo real de las pruebas en ejecución, incluyendo métricas de desempeño y resultados parciales de validación de mensajes.
- **Reportes:** centraliza la visualización de resultados finales y métricas agregadas, obtenidas desde el *backend*, permitiendo al usuario analizar los comportamientos de los flujos de prueba.

La Figura 3.2 muestra la arquitectura general de los módulos del *frontend*.

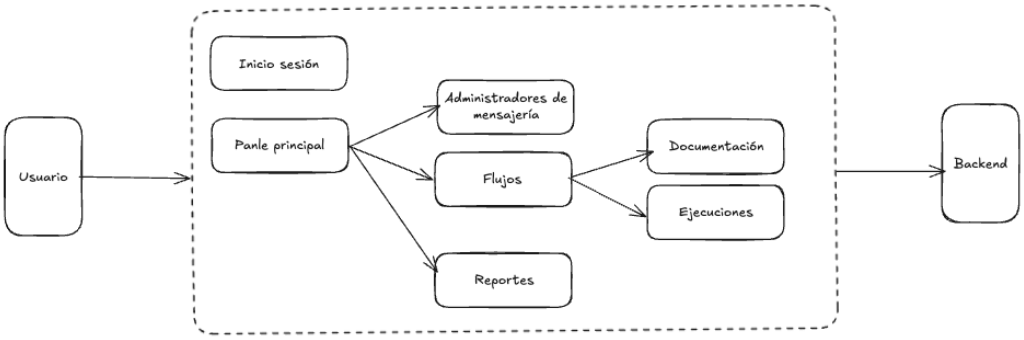


Figura 3.2: Arquitectura general de módulos del *frontend*

3.4.2. Tecnologías y herramientas utilizadas

La implementación del prototipo se realizó utilizando tecnologías que facilitaron el desarrollo modular, la comunicación asíncrona y la mantenibilidad del sistema, en coherencia con los principios de *Domain-Driven Design* (DDD) planteados por Vernon (2016). La selección de las herramientas se basó en tres criterios principales:

1. Compatibilidad con arquitecturas orientadas a eventos.
2. Soporte para un modelo de dominio estructurado y tipado.
3. Facilidad de integración con entornos de mensajería, observabilidad y almacenamiento distribuido.

3.4.2.1. *Backend*

El componente *backend*, se implementó en *Node.js* (v24.9.0) utilizando *TypeScript* (v5.8), lo que permitió la tipificación estática para garantizar robustez y mantenibilidad (Microsoft). Este entorno favoreció la implementación modular, la reutilización de componentes y la definición clara de contratos del dominio, en concordancia con el enfoque de diseño estratégico del DDD y las buenas prácticas de código limpio propuestas por Martin (2008). Se usaron un conjunto de librerías que facilitaron la comunicación entre servicios, la validación de datos y la instrumentación de métricas:

- **Express (v4.21.2):** *framework* HTTP ligero utilizado para la exposición de servicios REST, con *middlewares* personalizados para autenticación, validación y manejo de errores (Express.js Foundation).
- **amqplib (v0.10.4):** cliente oficial para el protocolo AMQP, empleado para la conexión, publicación y consumo de mensajes con RabbitMQ (SquareMo).
- **MongoDB (v6.18.0):** base de datos documental que almacenó la configuración de brokers, usuarios, flujos y resultados de pruebas, integrándose con los repositorios del dominio (MongoDB Inc.).
- **Socket.io (v4.8.1):** permitie la comunicación bidireccional en tiempo real entre el *frontend* y el *backend*, utilizada para el seguimiento de ejecuciones y flujos de prueba (Socket.IO Team).
- **Multer (v2.0.2):** maneja la carga de archivos, especialmente documentos AsyncAPI y flujos de prueba definidos por el usuario (Express.js Foundation).
- **js-yaml (v4.1.0):** procesa documentos YAML de especificaciones AsyncAPI, integrándose con el módulo *Documentation Reader* (Nodeca).

- **@asyncapi/parser (v3.4.0):** permitie interpretar y validar contratos de eventos definidos bajo el estándar AsyncAPI, garantizando la consistencia estructural de los mensajes (AsyncAPI Initiative).
- **@aws-sdk/client-s3 (v3.927.0):** integra la herramienta con *Cloudflare R2* para el almacenamiento distribuido de artefactos y resultados (Amazon Web Services).
- **bcrypt** y **jsonwebtoken:** gestionan la autenticación y la seguridad de usuarios mediante JWT, implementados en el módulo *Authenticator* (bcrypt Project Contributors; Auth0 Inc.).
- **class-validator** y **class-transformer:** aseguran la integridad y transformación coherente de las entidades del dominio en las interfaces de la API (typestack).
- **Swagger-jsdoc** y **Swagger-UI:** generan la documentación interactiva de la API (Swagger OpenAPI Initiative).
- **@edaniel30/loki-logger:** biblioteca personalizada desarrollada para la integración con *Grafana Loki*, utilizada en la recolección centralizada de logs, métricas y errores del sistema (Rivera).
- **dotenv** y **cors:** gestionan variables de entorno y políticas de acceso, facilitando la configuración en entornos distribuidos (dotenv Project; Express.js Foundation).

Para asegurar la calidad y consistencia del código, el entorno de desarrollo incorporó herramientas de automatización y control de estilo:

- **Jest (v29.7.0)** y **ts-jest:** ejecutan las pruebas unitarias y de integración, verificando la lógica de dominio y los casos de uso principales (Meta Open Source).
- **Prettier, Husky y Lint-staged:** garantizan un formato de código uniforme y la ejecución automática de validaciones previas a los *commits* (Prettier Core Team; Typicode; Okonetchnikov, Andrey).
- **Nodemon** y **ts-node:** habilitan la recarga en caliente durante el desarrollo, acelerando el ciclo de pruebas e iteraciones (Remy Sharp; TypeStrong).
- **rimraf:** automatiza la limpieza de compilaciones previas durante los procesos de construcción (*build*) (Isaac Z.).

3.4.2.2. *Frontend*

El componente *frontend*, desarrollado con *Angular* (v19.0.3), se diseñó bajo un enfoque modular que replica los contextos funcionales del *backend*. La interfaz se construyó sobre *Angular Material*, proporcionando un diseño limpio, responsivo y consistente con los principios de usabilidad moderna.
Las dependencias principales del *frontend* incluyeron:

- **@angular/core, @angular/router, @angular/forms:** núcleo del framework Angular, encargado de la gestión de componentes, rutas y formularios dinámicos (Google LLC and Angular Team).
- **@angular/material:** biblioteca de componentes visuales basada en las guías de diseño Material Design, empleada para la construcción del panel principal y los formularios de configuración (Google LLC and Angular Material Team).
- **rxjs (v7.8.0):** biblioteca para programación reactiva, utilizada en la gestión de flujos asincrónicos y la comunicación con el *backend* (ReactiveX Contributors).
- **SweetAlert2 (v11.26.3):** herramienta de interacción visual para la notificación de eventos, errores y confirmaciones al usuario (SweetAlert2 Team).

3.4.2.3. Integración y monitoreo

El sistema integró un conjunto de herramientas que fortalecieron la observabilidad, la automatización del desarrollo y la gestión del ciclo de vida del software, garantizando coherencia con las buenas prácticas de calidad y mantenibilidad propuestas por Martin (2008).

- Grafana Loki: centraliza los registros generados por el *backend*, facilitando el análisis de logs, la detección de errores y la visualización de métricas en tiempo real de la herramienta de pruebas.
- Swagger UI: documenta los endpoints REST expuestos por el *backend* y sirve como punto de verificación de la API, asegurando consistencia con el lenguaje ubicuo del dominio.
- GitHub: se utilizó como plataforma principal de control de versiones y repositorio del código fuente. Su adopción permitió mantener la trazabilidad de los cambios, y la gestión estructurada de versiones y ramas.

- GitHub Actions: se implementó como servicio de integración y entrega continua (CI/CD). A través de flujos automatizados de trabajo, la herramienta ejecutó pruebas unitarias, validaciones de construcción *(builds)* y análisis estáticos del código tras cada *commit* o *pull request*. Este proceso permitió detectar errores tempranos y asegurar que cada versión del sistema mantuviera un nivel de calidad constante antes de su integración a la rama principal.

El uso conjunto de estas herramientas garantiza la observabilidad operacional, la verificación automatizada del código y la trazabilidad de los procesos de desarrollo, fortaleciendo los atributos de confiabilidad, mantenibilidad y calidad estructural del software.

3.4.3. Proceso de desarrollo e implementación

El proceso de desarrollo del prototipo se llevó a cabo de forma iterativa e incremental, siguiendo una estrategia que permitió validar continuamente los avances técnicos y funcionales del sistema. El objetivo fue garantizar que cada módulo del sistema se implementara con alta cohesión, bajo acoplamiento y una clara correspondencia con los contextos delimitados definidos en el diseño arquitectónico.
Durante la implementación se estableció una estrategia de trabajo basada en tres ejes principales:

- **Desarrollo orientado al dominio:** Las entidades, agregados y casos de uso se diseñaron para reflejar los conceptos y procesos reales del dominio de las pruebas asincrónicas. Cada capa arquitectónica mantuvo su responsabilidad específica: el dominio encapsuló la lógica de negocio, los casos de uso orquestaron las interacciones, los proxies coordinaron operaciones distribuidas, y la infraestructura adaptó servicios externos mediante el patrón *Repository*.

- **Automatización y control de calidad:** Se aplicaron prácticas de integración continua, validación de código y ejecución de pruebas unitarias en cada módulo, utilizando flujos automatizados configurados en *GitHub Actions*. Se estableció un umbral mínimo de cobertura del 80 %, conforme al umbral comúnmente utilizado en los proyectos de software como indicador de calidad del conjunto de pruebas (Brandt (2025)). La validación del formato de código se automatizó mediante *Prettier* con *hooks* de *Husky* y *lint-staged*, garantizando consistencia estilística en cada *commit*.

- **Refinamiento progresivo del código:** Se realizaron ciclos continuos de refactorización y revisión de código, priorizando la legibilidad, simplicidad y mantenibilidad, en coherencia con los principios de código limpio planteados por Martin (2008). Se aplicaron sistemáticamente los principios SOLID *(Single Responsibility, Open/Clo-*

sed, Liskov Substitution, Interface Segregation y Dependency Inversion), con especial énfasis en el principio de responsabilidad única y la inversión de dependencias.

3.4.3.1. Desarrollo asistido por inteligencia artificial

Tanto el *frontend* como el *backend* se desarrollaron empleando entornos de desarrollo asistidos por inteligencia artificial (IA) mediante las herramientas Cursor IDE y ClaudeCode. Estas plataformas representan una evolución en los paradigmas de desarrollo de software, donde modelos de lenguaje de gran escala (LLMs) actúan como asistentes contextuales capaces de generar, analizar y refactorizar código en tiempo real. Este enfoque se alinea con las tendencias identificadas en el *State of AI-Assisted Software Development Report 2025*, el cual destaca que más del 90 % de los desarrolladores perciben mejoras en productividad, calidad del código y eficiencia general al integrar herramientas basadas en IA en sus flujos de trabajo (Google Cloud and DORA Research Program (2025)).

- **Cursor IDE** es un entorno de desarrollo integrado construido sobre Visual Studio Code, que incorpora modelos de IA (incluyendo GPT-4, Claude Sonnet) directamente en el flujo de trabajo del programador. Permite interacción mediante lenguaje natural para generación de código, refactorización, documentación y resolución de problemas, manteniendo consciencia del contexto completo del proyecto (Cursor Inc.).
- **Claude Code** es una herramienta de línea de comandos desarrollada por Anthropic que proporciona capacidades de desarrollo asistido mediante el modelo Claude (Sonnet 4.5). Ofrece funcionalidades avanzadas de exploración de código base, ejecución de comandos, manipulación de archivos y razonamiento sobre arquitecturas complejas, con capacidad para gestionar conversaciones extensas y mantener contexto de hasta 200,000 tokens (Anthropic).

Configuración de reglas de desarrollo y convenciones

Para estandarizar la interacción con los asistentes de IA y mantener uniformidad en el estilo de desarrollo, se definieron archivos de reglas específicos. Esto teniendo como base los hallazgos de Barke et al. (2022) y Klemmer et al. (2024), quienes destacan que la ausencia de convenciones claras en el uso de asistentes de IA puede derivar en inconsistencias arquitectónicas, divergencias de estilos y pérdida de trazabilidad del código.

- `.cursorrules`: Archivo de configuración específico para Cursor IDE que define las convenciones de desarrollo del proyecto. Este archivo es automáticamente leído por Cursor al iniciar cualquier sesión de desarrollo, instruyendo al modelo de IA sobre las reglas arquitectónicas, patrones de diseño y convenciones de código a seguir.

- `CLAUDE.md`: Documento de instrucciones para Claude Code que describe exhaustivamente la arquitectura del proyecto, entidades del dominio, flujos de datos, stack tecnológico, convenciones de nomenclatura, patrones de diseño aplicados y comandos útiles. Este archivo actúa como contexto persistente que guía al asistente en la generación de código consistente con la arquitectura establecida.

Estructura y contenido de los archivos de reglas

Los archivo de reglas y contexto se estructuraron siguiendo las recomendaciones sobre colaboración humano–IA planteadas por Treude (2023), quien propone la definición explícita de reglas, límites y contextos persistentes para asegurar la coherencia entre el conocimiento del modelo y la arquitectura del sistema. En coherencia con los principios de lenguaje ubicuo y modelado del dominio establecidos por Vernon (2016), el archivo actuó como un documento de alineación contextual.

1. **Descripción del proyecto**: Objetivos, características clave y propósito del sistema de pruebas asincrónicas.
2. **Arquitectura detallada**: Organización por capas (dominio, aplicación, proxy, infraestructura, entrada), con explicación de la ubicación de cada componente en la estructura de directorios y su responsabilidad específica.
3. **Patrones de diseño aplicados**: Documentación explícita de Repository, Builder, Factory, Proxy, Adapter, Observer y Dependency Injection, con ejemplos de implementación específicos del proyecto.
4. **Stack tecnológico**: Lista completa de dependencias con versiones exactas y justificación de cada elección tecnológica.
5. **Convenciones de código**: Reglas de nomenclatura (PascalCase para clases e interfaces, camelCase para métodos y funciones, UPPER_SNAKE_CASE para constantes), estructura de clases con patrón Factory, uso de decoradores de `class-validator` y `class-transformer`.
6. **Información específica del dominio**: Descripción detallada de cada entidad, sus propiedades, métodos de construcción y relaciones.
7. **Comandos útiles**: Scripts npm para desarrollo, testing, compilación, formateo y gestión de base de datos, facilitando operaciones repetitivas.
8. **Dependencias externas**: Configuración detallada de integraciones con MongoDB, RabbitMQ, Cloudflare R2, AsyncAPI Parser y Socket.io, incluyendo variables de entorno requeridas.

Beneficios del desarrollo asistido por IA en el proyecto

La integración de herramientas de IA en el proceso de desarrollo aportó beneficios cuantificables y cualitativos:

1. **Aceleración del desarrollo**: La generación asistida de código para estructuras repetitivas (repositorios MongoDB con mapeo dominio-modelo, handlers HTTP con validaciones, casos de uso con inyección de dependencias) redujo significativamente el tiempo de implementación de componentes estándar, permitiendo mayor foco en lógica de negocio compleja.

2. **Consistencia arquitectónica**: Al tener las convenciones documentadas, cada componente generado siguió automáticamente los patrones establecidos, evitando desviaciones arquitectónicas y deuda técnica.

3. **Refactorización asistida**: Cuando se identificaron mejoras arquitectónicas (como la introducción de la capa de Proxy para operaciones distribuidas), la IA facilitó la migración del código existente al nuevo patrón, manteniendo la funcionalidad mientras se mejoraba la estructura.

4. **Documentación continua**: Las interacciones con el asistente generaron documentación implícita del razonamiento detrás de decisiones técnicas, que posteriormente se formalizó en comentarios de código y documentación Swagger.

5. **Generación de pruebas unitarias**: La IA generó esqueletos de pruebas unitarias para casos de uso, con mocks de repositorios y validaciones de comportamiento, facilitando alcanzar la cobertura objetivo del 80 %.

6. **Mitigación de curva de aprendizaje**: Para tecnologías específicas como AsyncAPI Parser o Cloudflare R2 SDK, la IA proporcionó ejemplos de uso idiomáticos y mejores prácticas, reduciendo el tiempo de investigación de documentación externa.

Consideraciones éticas y limitaciones

Es importante destacar que, si bien las herramientas de IA aceleraron el desarrollo, el rol del desarrollador humano fue fundamental en:

- Diseñar la arquitectura general del sistema y definir los contextos delimitados.
- Establecer las convenciones y reglas documentadas.
- Validar la corrección lógica y seguridad del código generado.

- Tomar decisiones sobre arquitectónicos (ej. introducir capa de *Proxy* vs. manejar distribución en casos de uso).
- Garantizar la calidad mediante revisión de código y pruebas exhaustivas.

La IA actuó como herramienta amplificadora de productividad, no como sustituto del razonamiento ingenieril. El resultado fue un sistema donde cada componente refleja decisiones conscientes de diseño, validadas mediante múltiples ciclos de revisión.

3.4.4. Resumen de la fase

Durante esta fase se materializó la propuesta conceptual en un sistema funcional compuesto por dos componentes principales: un *backend* monolítico modular y un *frontend* independiente, ambos diseñados conforme a los principios del *Domain-Driven Design* (DDD) descritos por Vernon (2016).
El resultado de la implementación fue un prototipo funcional que permitie:

- Ejecutar flujos reales de mensajes sobre RabbitMQ.
- Monitorear y registrar métricas de desempeño.
- Visualizar resultados e identificar errores en tiempo real.

No se incluyó la integración con *pipelines* CI/CD empresariales ni el despliegue automatizado en la nube, los cuales se consideran extensiones posibles en trabajos futuros.
En términos metodológicos, la fase de prototipado permitió pasar de la abstracción del diseño a la implementación técnica del sistema, consolidando las funcionalidades esenciales identificadas como prioritarias: trazabilidad de eventos, validación de contratos, simulación de flujos y monitoreo de métricas de calidad.

Capítulo 4

Evaluación

En este capitulo se presenta la fase de Testear, la etapa final de la metodología *Design Thinking*, en la cual se evalúa el prototipo desarrollado con el propósito de validar su efectividad y su grado de alineación con las necesidades identificadas en las fases previas de Empatizar, Definir, Idear y Prototipar. En esta etapa se sometió la herramienta a un proceso de verificación sistemático en condiciones controladas, con el fin de determinar su capacidad para mejorar la trazabilidad de eventos, facilitar la detección temprana de errores y fortalecer la confiabilidad de los sistemas basados en arquitecturas orientadas a eventos.
Para lograrlo, se diseñó un proceso de evaluación que combinó pruebas funcionales y no funcionales, orientadas a medir el comportamiento de la herramienta frente a criterios de calidad del software como confiabilidad, rendimiento, trazabilidad y facilidad de uso.

4.1. Configuración del entorno de pruebas

Para garantizar la reproducibilidad y el control de las condiciones de validación, las pruebas del prototipo se realizaron en un entorno local configurado con componentes virtualizados mediante contenedores Docker. Este enfoque permitió aislar los servicios y replicar fácilmente las condiciones del experimento, siguiendo las recomendaciones sobre reproducibilidad en estudios empíricos de ingeniería de software propuestas por González-Barahona and Robles (2012).
Adicionalmente, se integró el servicio *Cloudflare R2* en su capa gratuita, utilizado como almacenamiento externo de la documentación Asyncapi de los flujos.
Esta configuración mixta (que combinó infraestructura local virtualizada con servicios en la nube de bajo costo) ofreció un entorno de evaluación controlado, reproducible y representativo de contextos reales de desarrollo de software distribuido.

4.1.1. Entorno de desarrollo local

El desarrollo y ejecución principal del prototipo se realizaron en un equipo *Mac Mini M4* con *macOS Thaoe 26.0.1*, con 16 GB de memoria RAM. Este entorno alojó los servicios de *frontend*, *backend* y la base de datos. El *backend* y el *frontend* se ejecutaron de manera independiente, utilizando puertos distintos en entorno local: el *backend* en el puerto 3000 y

el *frontend* en el puerto 4200. También se configuró un entorno complementario en un servidor doméstico con sistema operativo *Ubuntu* y 8 GB de memoria RAM. En este servidor se desplegaron los servicios de mensajería y observabilidad mediante contenedores Docker.

4.1.2. Infraestructura de servicios virtualizados

Para la simulación y monitoreo de los flujos de eventos se desplegaron los servicios de mensajería y observabilidad mediante contenedores Docker. Los servicios se organizaron de la siguiente forma:

- **RabbitMQ:** contenedor independiente basado en la versión 3.13.3, con el administrador de Rabbit habilitado y los siguientes complementos activos: `rabbitmq_mqtt`, `rabbitmq_web_mqtt`, `rabbitmq_amqp1_0` y `rabbitmq_event_exchange`. Los puertos expuestos fueron 5672 (mensajería AMQP) y 15672 (interfaz de administración web).
- **MongoDB:** desplegada en contenedor independiente (versión 8.0), con puerto expuesto 27017. Esta base de datos almacenó los registros de usuarios, configuraciones de flujos y resultados de las ejecuciones de prueba.
- **Grafana Loki:** configurada en un contenedor compuesto que incluyó los servicios de Alloy, Grafana y Nginx, junto con Loki. Este conjunto expuso el puerto 3100 para la recolección y consulta de logs generados por el *backend*. La integración con la biblioteca `@edaniel30/loki-logger` permitió el envío estructurado de registros de ejecución para su monitoreo en tiempo real.

4.2. Prubas y criterios de evaluación

4.2.1. Pruebas funcionales

Las pruebas funcionales se enfocaron en verificar el comportamiento esperado de los módulos principales del sistema: autenticación de usuarios, gestión de intermediarios de mensajería, administración de flujos de prueba y monitoreo de métricas. Cada módulo fue sometido a escenarios de validación definidos en función de los casos de uso especificados durante la fase de diseño.
Las pruebas se ejecutaron en el entorno local utilizando Jest como marco de pruebas unitarias, complementadas con pruebas manuales que simulaban eventos a través de RabbitMQ. Se emplearon mensajes estructurados en formato JSON, definidos según contratos de intercambio entre productores y consumidores, con el fin de evaluar la correcta validación de esquemas y el manejo de errores en la comunicación asíncrona.

4.2.2. Pruebas no funcionales

Las pruebas no funcionales se orientaron a evaluar la calidad estructural y el comportamiento del sistema bajo condiciones variables de carga, volumen de mensajes y concurrencia. Se emplearon métricas derivadas de las categorías propuestas por el Consortium for Information & Software Quality (CISQ) (2022) (confiabilidad, rendimiento, mantenibilidad y eficiencia de desempeño) adaptadas al contexto de arquitecturas orientadas a eventos.
Los principales indicadores medidos fueron:

- **Tiempo promedio de entrega de eventos:** intervalo entre la publicación y la recepción del mensaje por parte del consumidor.
- **Tasa de pérdida o duplicidad de mensajes:** proporción de mensajes no entregados o replicados durante la simulación.
- **Estabilidad del sistema:** número de ejecuciones exitosas consecutivas sin errores críticos.

Para la observación y registro de estas métricas se utilizaron los paneles de *Grafana*, configurados para recolectar información en tiempo real desde los contenedores de Docker mediante el módulo de monitoreo Loki. Adicionalmente, se analizaron los reportes de cobertura generados por `Jest` para evaluar la extensión de las pruebas unitarias e identificar posibles áreas del código no verificadas.

4.2.3. Criterios de evaluación

Con base en los objetivos específicos del proyecto, se definieron los siguientes criterios de evaluación:

1. **Cumplimiento funcional:** cada módulo debía ejecutar correctamente sus operaciones y mantener coherencia con los requisitos definidos.
2. **Confiabilidad:** el sistema debía mantener estabilidad durante múltiples ejecuciones de flujos asincrónicos sin pérdida de mensajes.
3. **Trazabilidad y observabilidad:** debía ser posible reconstruir el recorrido completo de un evento desde su emisión hasta su consumo, a través de logs y métricas visuales.
4. **Eficiencia:** los tiempos de entrega y procesamiento debían mantenerse dentro de los rangos esperados para entornos locales de prueba.
5. **Facilidad de uso:** la interfaz debía permitir configurar y ejecutar pruebas sin requerir conocimientos avanzados en administración de sistemas de mensajería.

En conjunto, la estrategia de evaluación permitió abordar de manera estructurada tanto los aspectos técnicos como operativos del sistema, ofreciendo una visión integral de su desempeño. El diseño de los casos de prueba y las métricas asociadas proporcionó evidencia cuantitativa y cualitativa que sustentó el análisis de resultados presentado en la siguiente sección.

4.3. Estrategia de evaluación

La estrategia de validación se diseñó con el propósito de demostrar empíricamente que la herramienta desarrollada resuelve los problemas identificados en las fases de Empatizar y Definir. Para ello, se adoptó un enfoque basado en casos de prueba técnicos automatizados que simulan escenarios reales de fallo y complejidad en sistemas distribuidos.
La validación se hizo con escenarios representativos de los problemas más frecuentes reportados por los desarrolladores: pérdida de mensajes, violación de contratos, falta de trazabilidad, problemas de orden y configuraciones erróneas de infraestructura. Cada escenario evalúa una o más de las necesidades priorizadas, permitiendo verificar la efectividad de la herramienta en condiciones controladas pero realistas.

4.3.1. Aplicación auxiliar para pruebas

Para validar la herramienta se implementó una aplicación auxiliar que simula componentes de un sistema distribuido real. Estos actúan como productores y consumidores de eventos, replicando comportamientos de microservicios en entornos de producción.
Componentes de la aplicación auxiliar de prueba:

1. Productores de eventos: Publican mensajes a *exchanges* de RabbitMQ. Estos productores simulan servicios que emiten eventos de negocio como órdenes creadas, pagos procesados, inventario actualizado, entre otros.

2. Consumidores de eventos: Se suscriben a colas específicas y procesan mensajes recibidos. Los consumidores, ejecutan lógica de negocio simulada y pueden generar eventos secundarios, replicando cadenas de procesamiento asíncrono.

3. Inyectores de fallos: Introducen condiciones de error controladas, como mensajes mal formados, duplicados, retrasos, pérdidas simuladas y violaciones de orden. Estos componentes permiten evaluar la capacidad de la herramienta para detectar anomalías.

Los componentes de esta aplicación auxiliar permiten simular escenarios de complejidad variable, desde flujos simples de un solo evento hasta cadenas complejas con múltiples productores y consumidores.

4.3.2. Escenarios de validación

Los escenarios de validación se diseñaron para cubrir sistemáticamente las necesidades priorizadas y los problemas más críticos identificados durante las fases de Empatizar y Definir. Cada escenario representa una situación real que los desarrolladores enfrentan al trabajar con sistemas basados en eventos, permitiendo evaluar la efectividad de la herramienta en condiciones representativas.
Se definieron seis escenarios de evaluación (EV1–EV6), cada uno orientado a validar una o más capacidades específicas de la herramienta. La Tabla 12 presenta una síntesis de los escenarios.

Tabla 12: Resumen de escenarios de validación

ID	Nombre	Necesidad relacionada	Objetivo
EV1	Validación de cumplimiento de contratos	Validación de contratos entre productores y consumidores	Detectar violaciones estructurales y semánticas en mensajes mediante validación contra esquemas *AsyncAPI*
EV2	Detección de eventos perdidos	Trazabilidad de eventos, Monitoreo de métricas	Identificar eventos esperados que no fueron entregados dentro de un flujo secuencial
EV3	Detección de mensajes duplicados	Monitoreo de métricas, Trazabilidad de eventos	Detectar cuando un mismo evento lógico es procesado más de una vez
EV4	Validación de orden de eventos	Trazabilidad de eventos, Simulación de escenarios reales	Validar que los eventos se procesen en la secuencia esperada en flujos con orden estricto
EV5	Validación de configuración de infraestructura	Simulación de escenarios reales, Estandarización del proceso	Detectar configuraciones inválidas de RabbitMQ antes de ejecutar las pruebas
EV6	Detección de eventos huérfanos	Validación de contratos, Detección de inconsistencias	Identificar eventos que llegan al broker pero no están definidos en la especificación *AsyncAPI*

4.3.2.1. Descripción de los escenarios

A continuación se presenta una descripción de cada escenario. Los detalles completos de configuración, casos de prueba y resultados esperados se documentan en el Anexo G.

EV1 – Validación de cumplimiento de contratos: Este escenario evalúa la capacidad de la herramienta para detectar violaciones de contrato en los mensajes intercambiados entre productores y consumidores. Se definió un esquema *AsyncAPI* con restricciones estrictas (campos obligatorios, tipos de datos, enumeraciones, patrones y rangos) y se ejecutaron ocho casos de prueba que incluyen mensajes válidos y mensajes con diferentes tipos de violaciones estructurales y semánticas.

EV2 – Detección de eventos perdidos: Se configuró un flujo secuencial de tres eventos (`OrderCreated` → `PaymentProcessed` → `OrderConfirmed`) para evaluar la detección de eventos que nunca llegan a su destino. Se simularon pérdidas en diferentes puntos del flujo, verificando que la herramienta identifique correctamente el último evento recibido y el evento faltante esperado.

EV3 – Detección de mensajes duplicados: Este escenario valida la capacidad de detectar procesamientos múltiples del mismo mensaje, un problema común causado por reintentos automáticos o configuraciones incorrectas de consumidores. Se ejecutaron casos con duplicaciones inmediatas, retardadas y múltiples, evaluando la precisión temporal del registro y el conteo de ocurrencias.

EV4 – Validación de orden de eventos: Se definió un flujo con orden estricto de cuatro eventos para evaluar la detección de violaciones de secuencia. Los casos de prueba incluyeron secuencias correctas, inversiones parciales y eventos que llegan antes de sus predecesores, verificando que la herramienta identifique la posición esperada versus la posición real de cada evento.

EV5 – Validación de configuración de infraestructura: Este escenario evalúa la capacidad preventiva de la herramienta para detectar configuraciones inválidas de RabbitMQ antes de ejecutar las pruebas. Se validaron casos de *exchanges* inexistentes, *exchanges* sin colas vinculadas y *bindings* con *routing keys* incorrectas.

EV6 – Detección de eventos huérfanos: Se configuró la herramienta para monitorear eventos que llegan al broker pero no están definidos en la especificación *AsyncAPI.* Este escenario permite identificar servicios *legacy* enviando eventos obsoletos, errores de configuración en productores o documentación desactualizada.

4.3.3. Criterios de evaluación y métricas

Para determinar el éxito de cada escenario de validación, se definieron métricas cuantitativas y cualitativas agrupadas en cuatro dimensiones de calidad, alineadas con los objetivos

del proyecto.

1. **Detección completa:** Mide la capacidad de la herramienta para identificar todos los casos de error o anomalía introducidos.
   - **Tasa de detección:** Porcentaje de casos problemáticos identificados correctamente del total de casos de prueba.
     - Fórmula: (Casos detectados/Total casos con error) $\times$ 100
     - Objetivo: $\geq 95\,\%$
   - **Falsos negativos:** Número de casos problemáticos no detectados.
     - Objetivo: 0 falsos negativos en escenarios críticos (EV1, EV2, EV5)
   - **Cobertura de casos de error:** Diversidad de tipos de error detectables.
     - Objetivo: 6 categorías de problemas detectables
     - Categorías: violación de contratos, eventos perdidos, mensajes duplicados, orden de eventos incorrecto, configuración de infraestructura inválida, eventos huérfanos no documentados
2. **Precisión de diagnóstico:** Evalúa la calidad de la información proporcionada sobre los problemas detectados.
   - **Especificidad del reporte:** Nivel de detalle en la descripción del problema.
     - Escala cualitativa:
       - *Básica:* Solo indica que ocurrió un error.
       - *Media:* Indica tipo de error y componente afectado.
       - *Alta:* Indica campo específico, valor esperado vs. recibido, contexto completo.
     - Objetivo: Nivel alto en todos los escenarios.
   - **Trazabilidad del error:** Capacidad de vincular el error a mensajes, *timestamps* y contexto específico.
     - Medida: Presencia de identificadores únicos, *timestamps* con precisión de milisegundos y contexto de correlación (routing key, exchange, payload).
     - Objetivo: 100 % de errores con trazabilidad completa.
   - **Claridad para desarrolladores:** Reporte claro, fácil de consumir sin necesidad de documentación adicional.
     - Evaluación cualitativa: Mensaje autoexplicativo y accionable.
     - Objetivo: Reportes comprensibles sin ambigüedad.

3. **Observabilidad** Evalúa la capacidad de capturar, registrar y monitorear eventos procesados durante las validaciones.
   - **Completitud de captura:** Porcentaje de eventos capturados durante la ventana de ejecución.
     - Medida: (Eventos capturados/Eventos publicados) × 100
     - Objetivo: 100 % de eventos capturados por spy queues.
   - **Preservación de metadatos:** Mantenimiento de información contextual de cada evento.
     - Medida: Presencia de routing key, exchange, headers y payload en eventos capturados.
     - Objetivo: 100 % de eventos con metadatos completos.
   - **Integración con observabilidad externa:** Disponibilidad de logs en sistemas como Grafana Loki.
     - Medida: Porcentaje de validaciones con logs registrados en Loki.
     - Objetivo: 100 % de validaciones con logs estructurados en Loki.
4. **Facilidad de configuración y uso** Mide la usabilidad general desde la perspectiva del desarrollador.
   - **Tiempo de configuración inicial:** Minutos necesarios para configurar un flujo desde AsyncAPI.
     - Objetivo: < 10 minutos para un flujo simple (3–5 eventos).
   - **Complejidad de definición de escenarios:** Número de pasos requeridos para definir y ejecutar un escenario.
     - Objetivo: ≤ 5 pasos (cargar AsyncAPI, crear flujo, crear ejecución, configurar validadores, ejecutar).

4.3.3.1. Matriz de relación: Necesidades → Escenarios → Métricas

La siguiente matriz relaciona las necesidades priorizadas con los escenarios de validación y con las métricas que permiten medir su cumplimiento, garantizando que cada necesidad sea validada de manera sistemática.

Tabla 13: Matriz de trazabilidad entre necesidades, escenarios y métricas

Necesidad	EVs	Métricas clave	Justificación

Detección de eventos perdidos y huérfanos	EV2, EV6	Tasa de detección (100 %), completitud de captura (100 %), integración Loki (100 %)	EV2 detecta eventos esperados que no llegaron; EV6 detecta eventos no documentados que sí llegaron.
Validación de contratos	EV1, EV6	Tasa de detección (100 %), especificidad del reporte (alta), trazabilidad del error (100 %)	EV1 valida contratos estructurales (JSON Schema); EV6 valida contratos semánticos (eventos permitidos).
Simulación de escenarios reales	EV2, EV3, EV4, EV5	Cobertura de casos de error (6 categorías), detección preventiva (EV5: 100 %)	Cubre eventos perdidos, duplicados, errores de orden, y configuración de infraestructura inválida.
Detección de anomalías operacionales	EV3, EV4	Falsos negativos (0 en EV3, EV4), precisión de posición (exacta), intervalos de duplicación (milisegundos)	EV3 detecta duplicados con timestamps precisos; EV4 detecta orden incorrecto con posición exacta.
Integración con ecosistema de desarrollo	Todos	Tiempo de configuración ($<$ 10 min), complejidad ($\leq$ 5 pasos), integración Loki (100 %)	Todos los EVs usan AsyncAPI como entrada, generan logs estructurados en Loki, y siguen flujo consistente.
Estandarización del proceso	Todos	Complejidad de definición ($\leq$ 5 pasos), claridad de reportes (autoexplicativos), preservación de metadatos (100 %)	Proceso reproducible: cargar AsyncAPI $\rightarrow$ crear flujo $\rightarrow$ crear ejecución $\rightarrow$ ejecutar $\rightarrow$ analizar resultados con contexto completo.

4.3.4. Protocolo de ejecución de escenarios

Para garantizar la reproducibilidad y consistencia en la validación, se definió un protocolo estándar aplicado de manera uniforme a todos los escenarios evaluados.

Fase 1: Preparación del entorno (pre-ejecución)

1. Iniciar los servicios de infraestructura (RabbitMQ, MongoDB, Loki) mediante *Docker Compose*.

2. Verificar la conectividad de RabbitMQ utilizando la herramienta.
3. Limpiar *py queues* de ejecuciones previas si es necesario.
4. Cargar la especificación AsyncAPI correspondiente al escenario en formato YAML.
5. Configurar y preparar productores externos que publicarán los eventos de prueba según el escenario.

Fase 2: Ejecución del escenario

1. Crear y configurar la ejecución desde la interfaz de la herramienta, seleccionando los validadores (EVs) a aplicar.
2. Iniciar la ejecución, lo cual activa la ventana de captura y los validadores configurados.
3. Los productores externos publican mensajes en el broker según la secuencia y timing definidos en el escenario.
4. Monitorear las notificaciones de validación en tiempo real vía WebSocket desde la interfaz.
5. Esperar la finalización automática de la ejecución al cumplirse la ventana de tiempo configurada.

Fase 3: Recolección de resultados (post-ejecución)

1. Extraer de la base de datos el estado final de la ejecución (`SUCCEEDED`, `FAILED`, `CANCELED`).
2. Recuperar los *test cases* con sus estados (`PASSED`, `FAILED`, `TIMEOUT`), *timestamps* y resultados detallados.
3. Consultar los logs estructurados en Grafana Loki filtrando por `executionId` y rango temporal.
4. Verificar el estado de la infraestructura en RabbitMQ Management UI (exchanges, bindings, mensajes en colas).
5. Capturar los resultados de ejecución y reportes generados desde la API o interfaz de la herramienta.

Fase 4: Análisis y validación

1. Comparar los resultados obtenidos contra los resultados esperados del escenario.
2. Verificar el cumplimiento de las métricas objetivo definidas para cada validador.
3. Documentar discrepancias, comportamientos inesperados o limitaciones observadas.
4. Registrar evidencia (logs de Loki, resultados de ejecución, capturas de interfaz).
5. Clasificar el resultado del escenario: *Exitoso* / *Parcial* / *Fallido*.

4.4. Limitaciones y supuestos del proceso de evaluación

Los escenarios de validación se diseñaron bajo condiciones controladas que permiten evaluar sistemáticamente la herramienta. No obstante, es fundamental establecer los supuestos y limitaciones bajo los cuales se ejecutará el proceso.

Supuestos

1. **Entorno controlado:** Las pruebas se ejecutan en un entorno local aislado, sin interferencias externas ni variabilidad propia de sistemas distribuidos reales.
2. **Aplicación auxiliar:** Los productores y consumidores utilizados representan comportamientos simplificados de microservicios reales y no contemplan lógica de negocio compleja.
3. **Volumen de datos moderado:** Los escenarios operan con hasta 10 eventos por caso. No se evalúa rendimiento bajo condiciones de carga extrema.
4. **Especificaciones AsyncAPI válidas:** Se asume que las especificaciones utilizadas son sintácticamente correctas y semánticamente coherentes.
5. **Infraestructura correctamente configurada:** RabbitMQ, MongoDB y Loki se encuentran instalados y configurados adecuadamente. No se contemplan fallos de infraestructura.
6. **Recursos disponibles:** El entorno de pruebas dispone de suficientes recursos (CPU, memoria y almacenamiento) para ejecutar todos los componentes sin degradación.

Limitaciones del alcance

1. **Broker único:** La validación se realiza exclusivamente con RabbitMQ. No se evalúa la integración con múltiples brokers ni con otros sistemas de mensajería (Kafka, ActiveMQ, AWS SQS).

2. **Sin participación de usuarios reales:** La evaluación se centra en pruebas técnicas automatizadas, sin incluir estudios de usabilidad con desarrolladores.

3. **Protocolos limitados:** Solo se evalúa comunicación basada en AMQP. Otros protocolos (MQTT, STOMP, HTTP webhooks) no forman parte del alcance.

4. **Complejidad moderada de *schemas*:** Los esquemas utilizados no incluyen validaciones condicionales avanzadas, referencias circulares ni modelos polimórficos complejos.

5. **Ausencia de latencia de red real:** Al ejecutarse localmente, los escenarios no capturan latencias variables, fallos de red o comportamientos distribuidos propios de entornos en la nube.

6. **No se evalúa seguridad:** No se consideran aspectos como autenticación avanzada, cifrado de mensajes o protección contra ataques maliciosos.

7. **Integración con CI/CD no evaluada:** Aunque la herramienta está diseñada para integrarse en pipelines, no se incluyen pruebas con Jenkins, GitLab CI o GitHub Actions.

8. **Escalabilidad horizontal no evaluada:** No se realizan pruebas con múltiples instancias de la herramienta ejecutándose en paralelo o en clúster.

9. **Ausencia de pruebas de carga y estrés:** El proceso de evaluación se centró en validar la efectividad funcional de la herramienta para detectar anomalías en flujos de eventos, sin incluir la caracterización de su rendimiento bajo condiciones de alta demanda. Esta decisión fue tomada siguiendo tres consideraciones principales:

    - El objetivo general del proyecto fue demostrar la viabilidad técnica de los mecanismos de validación propuestos. La efectividad de estos mecanismos es independiente del volumen de carga, dado que la lógica de validación opera de manera consistente sobre cada evento individual, sin verse afectada por el número total de mensajes procesados.
    - La ejecución rigurosa de pruebas de carga y estrés requiere infraestructura especializada como clústeres distribuidos, generadores de carga calibrados y sistemas

de monitoreo de recursos que exceden el alcance de este proyecto, orientado a la construcción y validación de un prototipo funcional.

- El entorno de evaluación empleado no representa condiciones de producción reales donde las métricas de rendimiento serían significativas y extrapolables. Reportar tiempos de respuesta, consumo de CPU o *throughput* obtenidos en este contexto podría generar conclusiones engañosas sobre el comportamiento esperado de la herramienta en entornos empresariales.

La instrumentación de métricas de rendimiento y la evaluación del sistema bajo condiciones de carga se identifican como líneas de trabajo futuro en la sección 5.5.

4.5. Resultados de la evaluación

Los resultados obtenidos de la ejecución de los escenarios de evaluación se organizaron en dos ejes: en el primero, se analizó el comportamiento de la herramienta frente a cada escenario individual, evaluando su capacidad para detectar anomalías específicas. En el segundo eje, se consolidaron métricas transversales que permitieron caracterizar el desempeño general del sistema en términos de confiabilidad, precisión y observabilidad. A continuación se presentan los hallazgos obtenidos en el proceso de validación, estructurados de acuerdo con los escenarios y las métricas definidas en la sección 4.3.3.

4.5.1. Material audiovisual complementario

Para facilitar la comprensión de las funcionalidades evaluadas, se elaboraron videos demostrativos que ilustran la ejecución de cada escenario de validación. Estos videos permiten observar el comportamiento de la herramienta en tiempo real y complementan los resultados documentados en esta sesión.
Los videos se encuentran disponibles en el siguiente enlace: repositorio de videos demostrativos.

4.5.2. EV1: Validación de cumplimiento de contrato

Se definió un contrato estricto para el evento `OrderCreated`, incluyendo campos obligatorios (`orderId`, `customerId`, `amount`, `currency`, `timestamp`), restricciones de tipos de datos, enumeraciones permitidas y patrones de formato. Se ejecutaron dos casos de prueba: (i) cinco mensajes válidos y (ii) cinco mensajes con violaciones intencionadas.

Resultados obtenidos

Tabla 14: Resultados de EV1 – Validación de cumplimiento de contratos

Ejecución	Procesados	Válidos	Inválidos	Estado	Tasa de detección
Messages OK	5	5	0	SUCCEEDED	100 % (5/5)
Messages with errors	5	2	3	FAILED	100 % (3/3)

En la segunda ejecución, la herramienta identificó correctamente las tres violaciones introducidas:

- **Campo obligatorio faltante (`currency`).** El mensaje con `orderId = 'ORD-12346'` omitió este campo, requerido por el esquema AsyncAPI. La herramienta reportó: *"Field 'currency' is required but missing"*.
- **Tipo de dato incorrecto (`amount` como cadena).** En el mensaje con `orderId = 'ORD-12347'`, el campo `amount` se envió como cadena: `"150.75"`. La herramienta reportó: *"Field '/amount' must be of type 'number'"*.
- **Valor fuera del dominio permitido (`currency`).** En el mensaje con `orderId = 'ORD-12349'` se envió `currency = "peso"`, valor no contenido en el enum {USD, EUR, COL, MXN}. La herramienta reportó:
  - *"Field '/currency' must be one of: [USD, EUR, COL, MXN]"*
  - *"Field '/currency' must match pattern: `"[A-Z]3`"*

Un hallazgo relevante fue el comportamiento ante campos adicionales no documentados. El mensaje con `orderId = 'ORD-12348'` incluyó `extraField`, no definido en el esquema. La herramienta lo marcó como **PASSED**, adoptando una política permisiva coherente con JSON Schema, donde las propiedades adicionales son válidas salvo que se declare explícitamente `additionalProperties: false`.
El reporte muestra por cada mensaje procesado:

- el *payload* en formato JSON,
- el estado de validación (`PASSED`/`FAILED`),
- los detalles específicos del error cuando aplicaba.

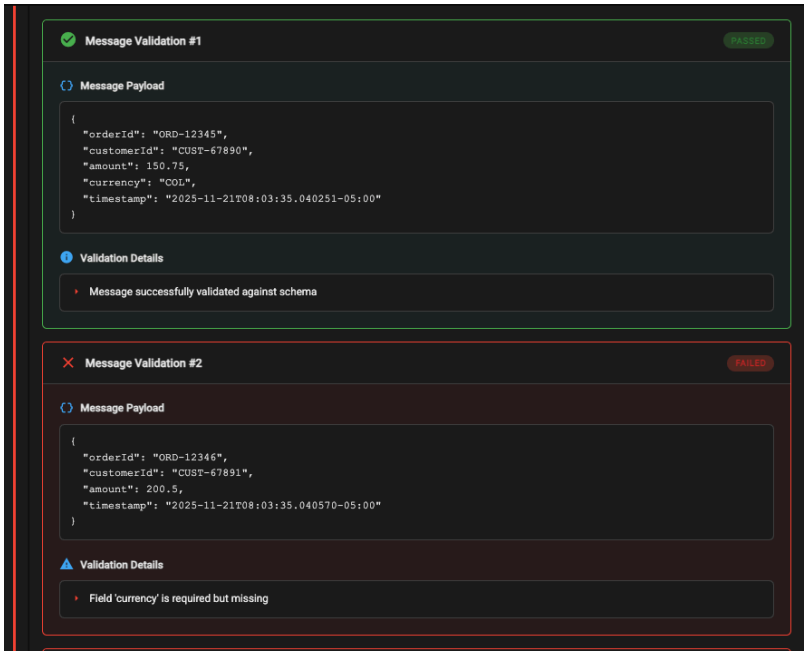


Figura 4.1: Resultados de EV1 - *Messages with errors*

Cumplimiento de métricas

- **Tasa de detección:** 100 % (3 de 3 violaciones detectadas)
- **Falsos negativos:** 0
- **Precisión del diagnóstico:** Alta, cada error incluye campo afectado, tipo de violación y valor esperado vs. recibido
- **Trazabilidad:** Completa, cada mensaje vinculado a *timestamp*, *routing key* y metadatos del evento

4.5.3. EV2: Detección de eventos perdidos

Se estableció un flujo compuesto por tres eventos consecutivos:

`OrderCreated` → `PaymentProcessed` → `OrderConfirmed`

Resultados obtenidos

Tabla 15: Resultados de EV2 – Detección de eventos perdidos

Caso de prueba	Eventos esperados	Eventos recibidos	Eventos faltantes	Estado	Ventana
OK	3	3	0	SUCCEEDED	50 ms
Loss in middle	3	1	2	FAILED	50 ms
Loss at end	3	2	1	FAILED	50 ms

En el caso *Loss in middle*, la herramienta detectó que únicamente el evento *OrderCreated* fue recibido. Para los eventos faltantes, el reporte incluyó:

- **Evento esperado:** `ev2.payment.processed`
- **Routing key esperado:** `ev2.payment.processed`
- **Estado:** No recibido
- **Tiempo transcurrido:** 50,011 ms (ventana completa de ejecución)
- **Exchange monitoreado:** `ev2.orders.events`
- **Eventos recibidos:** `ev2.orders.created`

Este reporte proporcionó el contexto necesario para identificar que el servicio responsable de publicar `PaymentProcessed` no ejecutó su lógica o falló antes de emitir el evento.

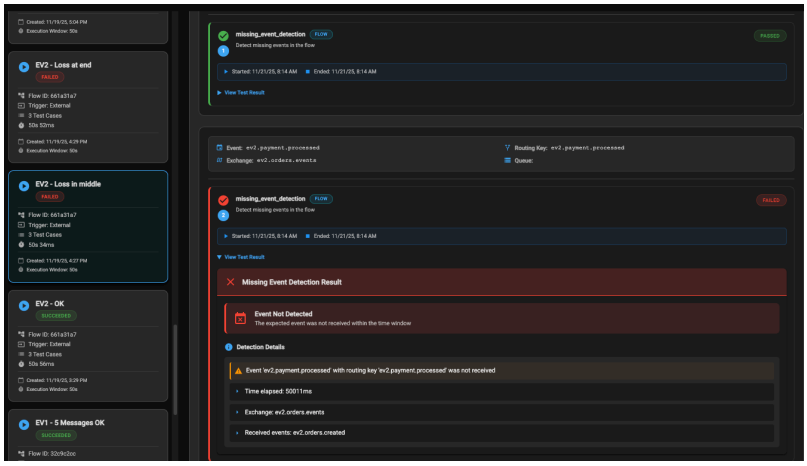


Figura 4.2: Resultados de EV2 - *Loss in middle*

En el caso *Loss at end*, la herramienta registró exitosamente `OrderCreated` y `PaymentProcessed`, pero detectó la ausencia de `OrderConfirmed`. El reporte indica que:

- *Event 'ev2.order.confirmed' with routing key 'ev2.order.confirmed' was not received*
- *Received events: ev2.orders.created, ev2.payment.processed*

Cumplimiento de métricas

- **Tasa de detección:** 100 % (2 escenarios con pérdidas correctamente identificados)
- **Completitud de captura:** 100 % de eventos publicados fueron detectados por las *spy queues*
- **Precisión de localización:** Exacta, identificación precisa del evento faltante dentro de la secuencia
- **Completitud de contexto:** Alta, registro de eventos previos facilita el análisis de causa raíz

4.5.4. EV3: Detección de mensajes duplicados

Se definió el evento `PaymentProcessed` con un identificador único `paymentId`. Se ejecutaron cuatro casos de prueba con diferentes patrones de duplicación:

- Sin duplicación.
- Duplicación inmediata (< 1 segundo).
- Duplicación retardada (5 segundos).
- Duplicaciones múltiples (5 ocurrencias).

Resultados obtenidos

Tabla 16: Resultados de EV3 – Detección de mensajes duplicados

Caso de prueba	**Payment ID**	**Ocurrencias**	**Intervalos (ms)**	**Estado**	**Detección**
No duplication	PAY-001	1	N/A	SUCCEEDED	Correcto (0 duplicados)
Immediate duplication	PAY-002	2	505	FAILED	Correcto (1 duplicado)

Delayed duplication	PAY-003	2	5,005	FAILED	Correcto (1 duplicado)
Multiple duplications	PAY-004	5	308, 306, 301, 306	FAILED	Correcto (4 duplicados)

Análisis detallado por caso

Caso sin duplicación. La herramienta procesó un único mensaje con `paymentId = "PAY-001"` y reportó:

- *'No duplicates detected'*
- *'Unique messages: 1'*
- *'Total messages processed: 1'*

Esto confirma que la herramienta no genera falsos positivos cuando los mensajes son únicos.

Caso de duplicación inmediata. El mensaje con `paymentId = "PAY-002"` fue procesado dos veces con un intervalo de 505 ms. La herramienta reportó:

- Total de ocurrencias: 2
- **Timestamps de ocurrencias**:
  - Primera: `2025-11-21T13:21:31.423Z`
  - Segunda: `2025-11-21T13:21:31.928Z`
- Intervalo calculado: 505 ms
- Estado: FAILED (duplicación detectada)

Caso de duplicación retardada. El mensaje con `paymentId = "PAY-003"` se procesó dos veces con un intervalo de 5,005 ms. Este caso validó que la detección de duplicaciones es independiente del tiempo, siempre que las ocurrencias se mantengan dentro de la ventana configurada.

Caso de duplicaciones múltiples. El mensaje con `paymentId = "PAY-004"` fue procesado cinco veces. La herramienta:

- Registró cinco timestamps individuales.
- Calculó cuatro intervalos consecutivos: 308, 306, 301, 306 ms.

- Identificó el primer y último timestamp del periodo de duplicación.
- Reportó:
  - *"Duplicate message detected with ID: PAY-004"*
  - *"Total occurrences: 5"*

Este caso demostró la capacidad de la herramienta para caracterizar patrones de duplicación masiva, típicos de problemas de idempotencia o reintentos fallidos.

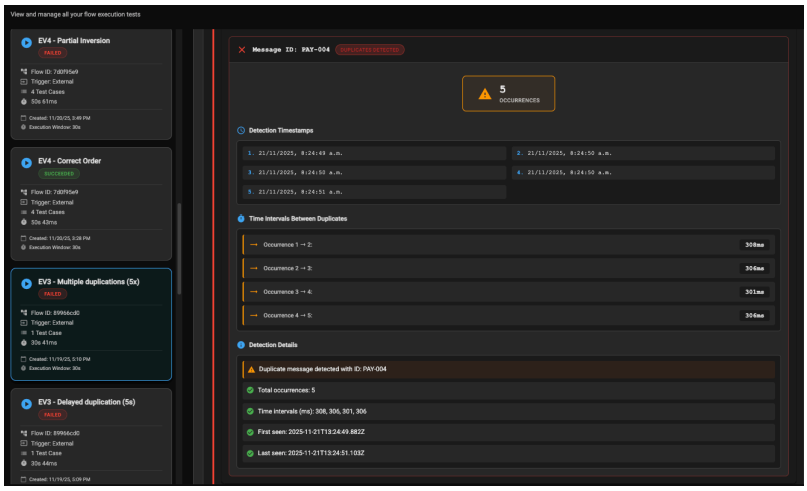


Figura 4.3: Resultados de EV3 - duplicaciones múltiples

La herramienta muestra *badges* destacados indicando el número de ocurrencias (ej.: `2 OCCURRENCES`, `5 OCCURRENCES`). En el caso de duplicaciones múltiples, se visualizó una tabla comparativa de intervalos, facilitando la identificación de patrones de error.

Cumplimiento de métricas

- **Tasa de detección:** 100 % (los tres casos con duplicación fueron detectados correctamente)
- **Granularidad temporal:** Precisión de milisegundos en todos los timestamps
- **Escalabilidad:** Detección efectiva de hasta cinco duplicaciones del mismo mensaje
- **Falsos positivos:** 0 (caso sin duplicación correctamente clasificado)

4.5.5. EV4: Validación de orden de eventos

Se definió un flujo estricto compuesto por cuatro eventos:

`OrderCreated (1)` → `PaymentProcessed (2)` → `OrderShipped (3)` → `OrderDelivered (4)`

Se ejecutaron tres casos de prueba:

- Secuencia correcta.
- Inversión parcial (eventos 2 y 3 intercambiados).
- Evento adelantado (evento 4 llega primero).

Resultados obtenidos

Tabla 17: Resultados de EV4 – Validación de orden de eventos

Caso de prueba	**Secuencia esperada**	**Secuencia recibida**	**Eventos PASSED**	**Eventos FAILED**	**Estado**
Correct Order	1→2→3→4	1→2→3→4	4	0	SUCCEEDED
Partial Inversion	1→2→3→4	1→3→2→4	2	2	FAILED
Early Delivery Event	1→2→3→4	4→1→2→3	0	4	FAILED

Análisis detallado por caso

Caso de orden correcto: Los cuatro eventos son publicados en la secuencia esperada (1→2→3→4). La herramienta validó cada evento contra su posición correspondiente, generando mensajes como:

- *'Event 'ev4.orders.created' arrived in correct position: 1*

Además, la herramienta construyó una secuencia acumulada paso a paso:

- Posición 1: `ev4.orders.created(1)`
- Posición 2: `ev4.orders.created(1) → ev4.payment.processed(2)`

- Posición 3: `... → ev4.orders.shipped(3)`
- Posición 4: Secuencia completa con los cuatro eventos

Caso de inversión parcial: La secuencia recibida fue 1→3→2→4, invirtiendo las posiciones de `PaymentProcessed` y `OrderShipped`. La herramienta detectó violaciones en ambos eventos:
Evento `ev4.payment.processed`:

- Posición esperada: 2
- Posición real: 3
- Estado: FAILED
- Mensaje: *Order violation detected for event 'ev4.payment.processed'*
- Secuencia hasta el momento: `1 → 3 → 2`

Evento `ev4.orders.shipped`:

- Posición esperada: 3
- Posición real: 2
- Estado: FAILED
- Mensaje: *Order violation detected for event 'ev4.orders.shipped'*

El reporte incluyó análisis adicional:

- *Arrived 1 position(s) early* (para `OrderShipped`)
- *Arrived 1 position(s) late* (para `PaymentProcessed`)

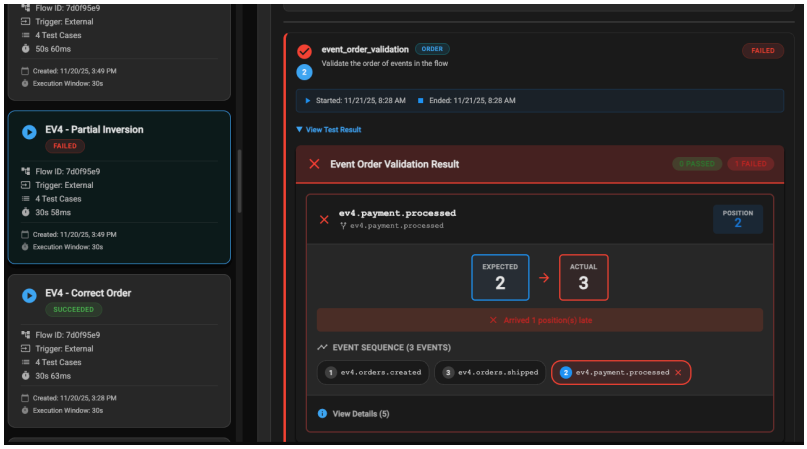


Figura 4.4: Resultados de EV4 - Caso de inversión parcial

Caso de evento adelantado: El evento `OrderDelivered` (posición 4) arribó primero, provocando un desalineamiento total del flujo. La herramienta detectó violaciones para los cuatro eventos:

- **ev4.orders.delivered** → esperado: 4, real: 1
- **ev4.orders.created** → esperado: 1, real: 2
- **ev4.payment.processed** → esperado: 2, real: 3
- **ev4.orders.shipped** → esperado: 3, real: 4

Este caso confirmó el comportamiento esperado del modo `strict`: una violación temprana invalida la secuencia completa.

Cumplimiento de métricas

- **Tasa de detección de violaciones:** 100 % (ambos casos incorrectos identificados correctamente)
- **Precisión de posición:** Exacta (posición esperada vs real para cada evento)
- **Claridad del reporte:** Alta (mensajes explícitos que describen el tipo de violación)
- **Contexto de secuencia:** Completo (se muestra la secuencia acumulada paso a paso)

4.5.6. EV5: Validación de configuración de infraestructura

Se definieron tres canales en la especificación *AsyncAPI*:

- `ev5.orders.created` — configuración válida.
- `ev5.inventory.check` — *exchange* inexistente: `ev5.inventory.events`.
- `ev5.orders.wrong.key` — *routing key* incorrecto.

Se ejecutaron dos casos de prueba:

- Configuración completamente válida.
- Caso con las tres validaciones fallidas.

Resultados obtenidos.

Tabla 18: Resultados de EV5 – Validación de configuración de infraestructura

Caso de prueba	**Pruebas ejecutadas**	**PASSED**	**FAILED**	**Problemas detectados**	**Estado final**
OK	1	1	0	Ninguno	SUCCEEDED
With Errors	3	1	2	*Exchange* no configurado, *Routing key* no enlazado	FAILED

Análisis detallado por caso

Prueba 1: `ev5.orders.created` (PASSED)

- *Exchange* esperado: `ev5.orders.events`
- *Exchange* configurado: `ev5.orders.events` ✓
- *Routing key* esperado: `ev5.orders.created`
- *Routing key* configurado: `ev5.orders.created` ✓

Resultado:

> *Infrastructure validation passed for the exchange ev5.orders.events*

Prueba 2: `ev5.inventory.check` (FAILED)

- Exchange esperado: `ev5.inventory.events`
- Exchange configurado: (vacío)
- Routing key esperado: `ev5.inventory.check`
- Routing key configurado: (vacío)

Problema detectado:

> *Exchange not configured on the broker*

Este caso representa un escenario frecuente donde la especificación *AsyncAPI* define un canal, pero el *exchange* correspondiente no existe en RabbitMQ. La herramienta detectó el problema de forma preventiva, evitando la pérdida silenciosa de mensajes al ejecutar las pruebas.

Prueba 3: `ev5.orders.cancelled` (FAILED)

- *Exchange* esperado: `ev5.orders.events`
- *Exchange* configurado: `ev5.orders.events` ✓
- *Routing key* esperado: `ev5.orders.wrong.key`
- *Routing key* configurado: (vacío)

Problema detectado:

> *Routing key not configured for the exchange ev5.orders.events"*

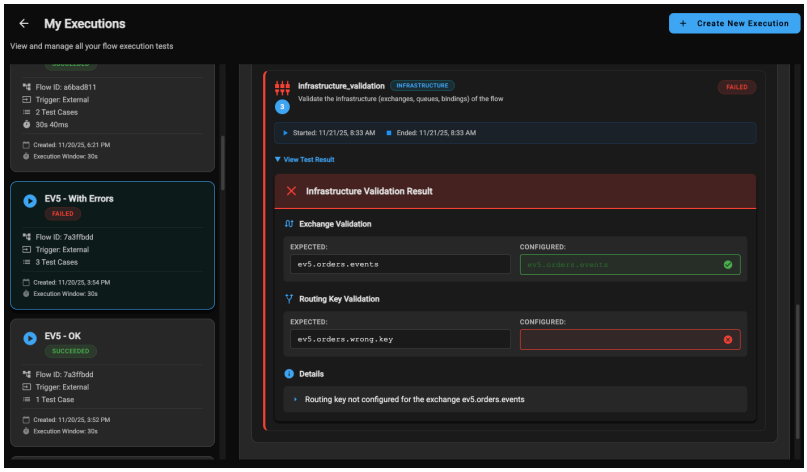


Figura 4.5: Resultados de EV5 - Error *exchange* sin *binding*

Este caso corresponde a una configuración parcial donde el *exchange* existe, pero no tiene un *binding*. La validación previene que los mensajes publicados sean descartados por el administrador de mensajes al no existir una ruta válida.

Cumplimiento de métricas

- **Cobertura de validación:** Validación completa de exchanges y bindings definidos en AsyncAPI.
- **Detección preventiva:** 100 % de configuraciones inválidas detectadas antes de ejecutar las pruebas.
- **Tiempo de validación:** $<$ 1 segundo para 3 canales.
- **Claridad del diagnóstico:** Alta — mensajes específicos indicando el componente afectado y el error encontrado.

4.5.7. EV6: Detección de eventos huérfanos

Se definieron dos eventos documentados en *AsyncAPI*:

- `order.created`
- `payment.processed`

La herramienta fue configurada para monitorear el exchange `ev6.orders.events` utilizando un *wildcard routing key* que captura todos los mensajes. Se ejecutaron cuatro casos de prueba:

- Solo eventos documentados.
- Un evento huérfano (`order.cancelled`).
- Eventos mixtos (documentados + huérfanos).
- Múltiples huérfanos.

Resultados obtenidos.

Tabla 19: Resultados de EV6 – Detección de eventos huérfanos

Caso de prueba	**Eventos documentados**	**Eventos huérfanos**	***Routing keys* huérfanos detectados**	**Estado**
Documented Events	2	0	Ninguno	SUCCEEDED
Single Orphan Event	0	1	order.cancelled	FAILED
Mixed Events	2	1	order.updated	FAILED
Multiple Orphans	0	2	order.cancelled, order.updated	FAILED

Análisis detallado por caso

Caso con eventos documentados: La herramienta procesó únicamente `order.created` y `payment.processed`, ambos definidos en *AsyncAPI*. Reporte:

- *No orphan events detected*
- Badge visual: `0 ORPHAN EVENTS` (✓ verde)
- Ambos test cases: PASSED

Esto confirmó la ausencia de falsos positivos.

Caso con un evento huérfano: Llega un mensaje con `rk = order.cancelled`, no definido en *AsyncAPI*. La herramienta generó un reporte detallado:

- Número de huérfanos: 1
- *Routing key* detectado: `order.cancelled`
- *Exchange:* `ev6.orders.events`

- *Queue* de captura: `spy.orphan.event`
- **Mesaje completo:**

```
{
  "orderId": "ORD-60002",
  "reason": "Customer request",
  "cancelledBy": "CUST-90002",
  "timestamp": "2025-11-21T08:35:31.730223-05:00"
}
```

Este nivel de detalle permite determinar si el evento proviene de:

1. Un servicio obsoleto que continúa emitiendo eventos obsoletos.
2. Un productor mal configurado.
3. Una especificación *AsyncAPI* desactualizada.

Caso con eventos mixtos. Incluyó eventos documentados (`order.created`, `payment.processed`) junto con un huérfano (`order.updated`). La herramienta:

- Se validó correctamente los eventos documentados.
- Se detectó el evento `order.updated` como huérfano.
- Se presentó el mensaje completo.

Esto demostró su capacidad para diferenciar entre eventos esperados y no esperados en la misma ejecución.

Caso con múltiples huérfanos. Se detectaron dos eventos huérfanos:

- `order.cancelled`
- `order.updated`

El reporte consolidado incluyó:

- **Resumen:** *2 orphan event(s) detected - not documented in AsyncAPI specification*
- **Detalles individuales:** *routing key, exchange* y mensaje completo para cada huérfano.

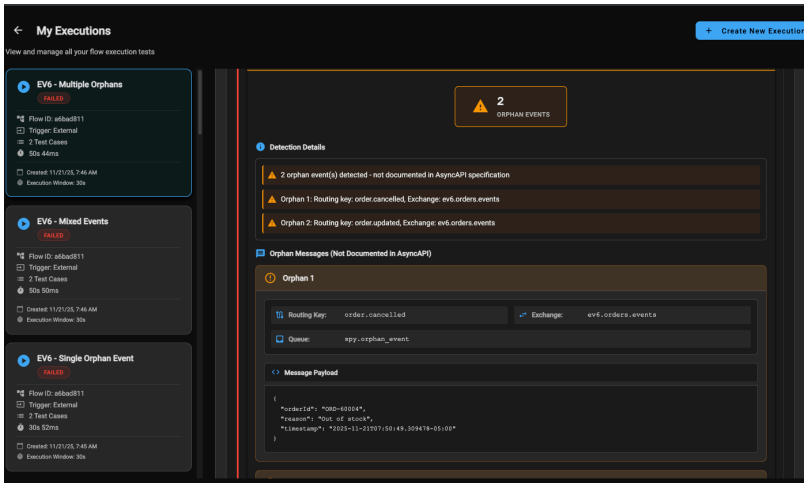


Figura 4.6: Resultados de EV6 - Eventos huerfanos detectados

Cumplimiento de métricas

- **Tasa de detección de huérfanos:** 100 % (todos los eventos no documentados fueron identificados).
- **Cobertura de monitoreo:** Todos los exchanges relevantes fueron monitoreados mediante wildcard.
- **Especificidad de reporte:** Alta — cada huérfano incluyó routing key, exchange y payload completo.
- **Falsos positivos:** 0 — ningún evento documentado fue clasificado como huérfano.

4.6. Análisis consolidado

A partir de los resultados individuales de cada escenario, se consolidaron métricas transversales que permitieron caracterizar el desempeño global de la herramienta.

4.6.1. Tasa de detección global

Tabla 20: Tasa de detección consolidada por escenario

Escenario	Casos con anomalías	Anomalías introducidas	Anomalías detectadas	Tasa de detección	Falsos negativos
EV1 - Contratos	1	3	3	100 %	0
EV2 - Eventos perdidos	2	3	3	100 %	0
EV3 - Duplicados	3	3	3	100 %	0
EV4 - Orden	2	6	6	100 %	0
EV5 - Infraestructura	1	2	2	100 %	0
EV6 - Huérfanos	3	4	4	100 %	0
TOTAL	**12**	**21**	**21**	**100 %**	**0**

La herramienta alcanzó una **tasa de detección del 100 %** en los seis escenarios evaluados, identificando correctamente las 21 anomalías introducidas sin generar falsos negativos. Este resultado valida la efectividad de los mecanismos de validación implementados y confirma el cumplimiento del objetivo establecido en las métricas de éxito.

4.6.2. Precisión de diagnóstico

Se evaluó la calidad de la información proporcionada por la herramienta al detectar anomalías, clasificando los reportes en tres niveles:

- **Nivel básico:** Indica únicamente que ocurrió un error.
- **Nivel medio:** Indica el tipo de error y el componente afectado.
- **Nivel alto:** Indica el campo específico, valor esperado vs recibido y contexto completo.

Tabla 21: Precisión de diagnóstico por escenario

Escenario	Nivel de detalle alcanzado	Elementos incluidos en el reporte
EV1 - Contratos	**Alto**	Campo afectado, tipo de violación, valor esperado vs recibido, payload completo

EV2 - Eventos perdidos	**Alto**	Evento esperado, routing key, tiempo transcurrido, eventos previos recibidos, exchange
EV3 - Duplicados	**Alto**	Identificador único, timestamps con precisión de milisegundos, intervalos entre ocurrencias
EV4 - Orden	**Alto**	Posición esperada vs real, secuencia acumulada, magnitud de la desviación
EV5 - Infraestructura	**Alto**	Exchange esperado vs configurado, routing key esperado vs configurado, tipo de problema
EV6 - Huérfanos	**Alto**	Routing key, exchange, queue de captura, payload completo, headers

La herramienta alcanzó consistentemente un nivel alto de precisión de diagnóstico en todos los escenarios evaluados. Los reportes generados incluyeron información contextual suficiente para que un desarrollador pudiera:

1. Comprender la naturaleza exacta del problema sin consultar documentación adicional.
2. Identificar el componente específico que debe corregirse.
3. Reproducir localmente las condiciones del fallo.
4. Implementar una solución dirigida sin necesidad de investigación exploratoria.

4.6.3. Observabilidad alcanzada

La observabilidad se evaluó en tres aspectos: captura de eventos, preservación de metadatos e integración con sistemas externos de monitoreo.

Captura: Se verificó que la herramienta capturara todos los eventos publicados durante las ventanas de ejecución. Los resultados consolidados de las 17 ejecuciones mostraron:

- **Eventos publicados totales:** 67
- **Eventos capturados por *spy queues:*** 67
- **Tasa de captura:** 100 %

No se observó pérdida de eventos durante el proceso de monitoreo, confirmando que el mecanismo de *spy queues* implementado con *bindings* a nivel de *exchange* funciona de manera completamente confiable.

Preservación de metadatos: Cada evento capturado incluyó consistentemente los siguientes metadatos:

- ***Routing key:*** Identificador del tipo de evento.
- ***Exchange:*** Origen del mensaje.
- ***Headers:*** Metadatos adicionales del mensaje (si existen).
- ***Payload:*** Contenido completo del mensaje en formato JSON.
- ***Timestamp:*** Momento de captura con precisión de milisegundos.
- ***Queue:*** Cola *spy* responsable de la captura.

La preservación completa de metadatos permitió realizar análisis retroactivos detallados y facilitó la correlación de eventos entre diferentes componentes del sistema.

Cumplimiento de métricas de observabilidad.

- **Completitud de captura:** 100 % (67/67 eventos capturados).
- **Preservación de metadatos:** 100 % (todos los eventos capturados con metadatos completos).

4.6.4. Facilidad de configuración y uso

Se evaluó la usabilidad de la herramienta midiendo el tiempo requerido para configurar y ejecutar un flujo de pruebas, así como la complejidad percibida del proceso.

Tiempo de configuración inicial: Se cronometró el tiempo necesario para configurar cada escenario desde cero:

1. **Creación del flujo:** < 2 minutos.
2. **Carga documentación *AsyncAPI:*** < 1 minuto.
3. **Configuración de ejecución (ventana, timeout, validadores):** < 3 minutos.
4. **Ejecución de pruebas:** Variable según escenario (30–50 segundos por ejecución).
5. **Análisis de resultados:** < 5 minutos.

Tiempo promedio total: < 10 **minutos** para configurar y ejecutar un flujo simple con 3–5 eventos.
Este resultado cumple la métrica objetivo establecida (< 10 minutos) y confirma que la herramienta puede integrarse eficientemente en flujos de trabajo de desarrollo sin causar interrupciones significativas.

Complejidad de definición de escenarios: El proceso de configuración siguió un flujo consistente compuesto por cinco pasos:

1. Cargar especificación *AsyncAPI* (archivo YAML).
2. Crear flujo desde la interfaz.
3. Crear ejecución asociada al flujo.
4. Seleccionar validaciones.
5. Ejecutar.

Este proceso cumple la métrica objetivo (<= 5 pasos) y demostró ser suficientemente intuitivo para que usuarios con conocimientos básicos en arquitecturas orientadas a eventos puedan utilizar la herramienta sin necesidad de capacitación extensa.

Oportunidades de mejora:

- **Exportación de reportes:** La falta de una funcionalidad para exportar resultados en formatos estándar (PDF, JSON) limita su uso en auditorías o documentación formal.
- **Comparación de ejecuciones:** La interfaz no permite una comparación visual lado a lado entre ejecuciones del mismo flujo.
- **Sugerencias de corrección:** Los reportes podrían enriquecerse con recomendaciones automáticas basadas en el tipo de error detectado.

4.6.5. Cumplimiento de necesidades priorizadas

La Tabla 22 establece la relación entre las necesidades priorizadas y los escenarios de validación ejecutados, evaluando el grado de cumplimiento alcanzado.

Tabla 22: Matriz de cumplimiento de necesidades priorizadas

Necesidad priorizada	Escenarios relacionados	Evidencia de cumplimiento	Nivel de cumplimiento
Trazabilidad de eventos	EV2, EV6	Detección 100 % de eventos perdidos y huérfanos; captura completa de metadatos *(routing key, exchange, payload, timestamp)*	**Completo**
Validación de contratos	EV1, EV6	Detección 100 % de violaciones estructurales y semánticas, diferenciación correcta entre eventos documentados y no documentados	**Completo**
Simulación de escenarios reales	EV2, EV3, EV4, EV5	Reproducción exitosa de pérdidas, duplicaciones, violaciones de orden y errores de configuración bajo condiciones controladas	**Completo**
Monitoreo de métricas de calidad	Todos los EVs	Registro de *timestamps* con precisión de milisegundos; cálculo de intervalos; métricas de detección y precisión	**Completo**
Integración con ecosistema de desarrollo	Todos los EVs	Interfaz web accesible, carga de *AsyncAPI* estándar, logs estructurados en Loki, tiempo de configuración $<$ 10 min	**Completo**
Estandarización del proceso de prueba	Todos los EVs	Flujo consistente de 5 pasos; reportes estructurados con formato uniforme; documentación mediante *AsyncAPI*	**Completo**

Cada necesidad fue abordada mediante uno o más escenarios de validación, y los resultados empíricos confirmaron que las funcionalidades implementadas responden de manera efectiva a los problemas identificados en las fases de *Empatizar* y *Definir*.

Capítulo 5

Conclusiones

La presente investigación tuvo como propósito desarrollar una herramienta de pruebas automatizadas que abordara las limitaciones identificadas en la validación de arquitecturas orientadas a eventos, específicamente aquellas que utilizan RabbitMQ como intermediario de mensajes. Los resultados obtenidos durante la fase de evaluación confirmaron la efectividad de la solución propuesta para mejorar la trazabilidad de eventos, la detección temprana de errores y la confiabilidad de sistemas distribuidos asíncronos. A continuación se presentan las conclusiones estructuradas conforme a los objetivos establecidos en el capítulo 1, se analizan las fortalezas y limitaciones evidenciadas, se establece la relación con la literatura existente y se discuten las implicaciones prácticas de los hallazgos.

5.1. Cumplimiento de objetivos

5.1.1. Objetivo general

El objetivo general del proyecto fue desarrollar una herramienta de pruebas automatizadas que integrara capacidades de monitoreo, validación y depuración para arquitecturas orientadas a eventos basadas en RabbitMQ. Este objetivo se cumplió mediante la implementación de un prototipo funcional capaz de:

- Monitorear flujos de eventos en tiempo real mediante *spy queues* configuradas a nivel de *exchange*.
- Validar contratos de mensajes utilizando especificaciones AsyncAPI y JSON Schema.
- Detectar anomalías temporales como eventos perdidos, duplicados y fuera de orden.
- Generar reportes estructurados con información contextual detallada.
- Integrarse con sistemas externos de observabilidad (Grafana Loki) para auditoría y análisis retroactivo.

La herramienta alcanzó una tasa de detección del **100 %** en los seis escenarios evaluados, identificando correctamente 21 anomalías sin falsos negativos. Este resultado supera el umbral objetivo establecido ($\geq 95\,\%$) y confirma la viabilidad técnica de la solución propuesta.

5.1.2. Objetivos específicos

Objetivo específico 1: Analizar las limitaciones de las metodologías y herramientas de pruebas existentes aplicadas a arquitecturas orientadas a eventos

Este objetivo se cumplió mediante el análisis sistemático de trabajos previos (CCTS, BOSET, AutoQUEST) y herramientas de uso industrial (Postman, JMeter), documentado en las secciones 2.2 y 3.2.3.
Donde se identificaron brechas relevantes:

- Ausencia de soporte nativo para protocolos de mensajería asíncrona (AMQP).
- Complejidad de despliegue y baja adopción industrial de prototipos académicos.
- Falta de mecanismos integrados para trazabilidad punto a punto de eventos.
- Dependencia de instrumentación manual o extensiones externas.

Estos hallazgos orientaron las decisiones de diseño, priorizando simplicidad de despliegue, integración con estándares (AsyncAPI) y automatización de la captura de eventos.

Objetivo específico 2: Diseñar la arquitectura y los componentes de una herramienta de pruebas automatizadas que integre mecanismos de trazabilidad, validación y depuración de eventos

El diseño arquitectónico se desarrolló bajo principios de *Domain-Driven Design* Vernon (2016), resultando en un monolito modular con separación clara de responsabilidades entre las capas de dominio, aplicación e infraestructura. La arquitectura modular facilita la extensibilidad y mantenibilidad, permitiendo agregar nuevos validadores sin alterar componentes existentes.

Objetivo específico 3: Implementar la herramienta propuesta siguiendo principios de ingeniería de software y buenas prácticas de desarrollo para entornos distribuidos asíncronos

La implementación se realizó en Node.js con TypeScript, aplicando principios SOLID y patrones como Repository, Factory y Adapter. Se establecieron mecanismos de calidad automatizados:

- **Cobertura de código:** 80 % mediante pruebas unitarias con Jest.
- **Integración continua:** GitHub Actions para validación automática por commit.
- **Control de estilo:** Prettier, Husky y lint-staged.
- **Desarrollo asistido por IA:** Uso de Cursor IDE y Claude Code con reglas estandarizadas.

El proceso mostró que el uso guiado de IA puede acelerar la implementación sin comprometer la calidad del código.

Objetivo específico 4: Definir métricas e indicadores que permitan evaluar la efectividad de la herramienta en términos de cobertura, confiabilidad y facilidad de uso

Se definieron métricas alineadas con los criterios de calidad propuestos por Consortium for Information & Software Quality (CISQ) (2022), organizadas en cuatro dimensiones:

- **Detección completa:** Tasa de detección, falsos negativos, cobertura de casos de error.
- **Precisión de diagnóstico:** Especificidad del reporte, trazabilidad del error, claridad para desarrolladores.

- **Observabilidad:** Completitud de captura, preservación de metadatos, integración con sistemas externos.
- **Facilidad de uso:** Tiempo de configuración, complejidad de definición de escenarios.

Estas métricas proporcionaron un marco objetivo para evaluar el desempeño del prototipo y compararlo con alternativas existentes.

Objetivo específico 5: Validar la herramienta mediante pruebas funcionales y no funcionales en un entorno controlado que reproduzca escenarios reales de operación

La validación se realizó mediante seis escenarios que representan problemas recurrentes en sistemas distribuidos:

- Violaciones de contrato.
- Eventos perdidos.
- Mensajes duplicados.
- Orden incorrecto.
- Configuración de infraestructura inválida.
- Eventos huérfanos.

Los resultados consolidados mostraron:

- **Tasa de detección global:** 100 % (21/21 anomalías detectadas).
- **Precisión de diagnóstico:** Alta en todos los escenarios.
- **Observabilidad:** 100 % de eventos capturados; 796 logs estructurados generados.
- **Tiempo de configuración:** < 10 minutos para flujos simples.

Estos resultados confirman que la herramienta cumple los objetivos funcionales y no funcionales establecidos, proporcionando una solución viable para el aseguramiento de calidad en arquitecturas orientadas a eventos.

5.2. Fortalezas evidenciadas

Cobertura integral

La herramienta validó exitosamente aspectos estructurales (contratos, infraestructura), temporales (orden, duplicación, pérdidas) y semánticos (eventos huérfanos) de los flujos asincrónicos, cubriendo las dimensiones críticas identificadas en la literatura reciente (Wu et al. (2022); Silva et al. (2025)). Esta cobertura integral diferencia la solución propuesta de herramientas genéricas como Postman o JMeter, que se enfocan principalmente en comunicación síncrona basada en HTTP.

Detección preventiva

La validación de infraestructura (EV5) permitió identificar configuraciones inválidas antes de ejecutar las pruebas, evitando pérdida silenciosa de mensajes y reduciendo significativamente el tiempo de depuración. Este enfoque de *shift-left testing* (detección temprana) es coherente con las mejores prácticas documentadas por Boehm and Basili (2001), quienes demostraron que el costo de corregir un defecto aumenta exponencialmente en fases posteriores del ciclo de desarrollo.

Independencia de plataforma

El uso de AsyncAPI como estándar de definición de contratos garantiza que la herramienta pueda aplicarse en sistemas implementados en diversos lenguajes de programación y frameworks. Este enfoque mantiene coherencia con los principios de diseño tolerante a fallos planteados por Vernon (2016), permitiendo que la solución sea portable y adaptable sin modificaciones estructurales.

5.3. Limitaciones identificadas

A pesar de los resultados positivos, se identificaron limitaciones técnicas que deben considerarse al interpretar los hallazgos y planificar extensiones futuras.

Validación evento por evento

Los validadores procesan eventos de forma individual sin rastrear flujos completos *end-to-end* mediante *correlation IDs*. Esta limitación impide reconstruir trazas de transacciones distribuidas que atraviesan múltiples servicios y eventos, lo cual sería especialmente valioso en escenarios complejos como *SAGAs* de larga duración. La implementación de trazabilidad distribuida requeriría instrumentación adicional y coordinación con servicios externos,

incrementando la complejidad del sistema.

Dependencia de configuración correcta

La validación de orden de eventos (EV4) depende de que los *bindings* entre exchanges y *spy queues* estén configurados correctamente. Configuraciones incompletas pueden impedir la captura de eventos necesarios, generando falsos positivos en la detección de eventos perdidos. Esta dependencia introduce fragilidad operativa y requiere que los usuarios comprendan la topología de RabbitMQ para configurar adecuadamente la herramienta.

Alcance limitado a RabbitMQ

La solución desarrollada se enfoca exclusivamente en RabbitMQ como intermediario de mensajes. Si bien esta especialización facilita una integración profunda con capacidades específicas del broker —como *spy queues*, *bindings* dinámicos y la Management API—, limita la aplicabilidad a sistemas que utilizan otros intermediarios como Apache Kafka, AWS SQS o Google Cloud Pub/Sub. La extensión a múltiples brokers requeriría una abstracción de las operaciones de mensajería y adaptadores específicos por plataforma.

Validación en entornos standalone

La evaluación de la herramienta se realizó contra una instancia única de RabbitMQ, sin considerar configuraciones de clúster. Según el teorema CAP (Gilbert and Lynch (2002); Brewer (2012)), el comportamiento de los sistemas distribuidos varía significativamente según se priorice consistencia o disponibilidad ante particiones de red. La extensión de la herramienta hacia entornos de producción con RabbitMQ en clúster requeriría validar su comportamiento bajo diferentes modos de replicación y documentar los *trade-offs* esperados según la configuración empleada.

5.4. Implicaciones prácticas

Los resultados sugieren que la adopción de herramientas especializadas para pruebas en arquitecturas orientadas a eventos puede:

- **Reducir el tiempo de depuración:** La detección temprana de errores mediante validación automatizada previene que defectos lleguen a producción, reduciendo el costo de corrección (Boehm and Basili (2001)).
- **Mejorar la confiabilidad del sistema:** La validación de contratos, orden y duplicación fortalece la coherencia del sistema distribuido y disminuye la probabilidad de estados inconsistentes.

- **Facilitar la transferencia de conocimiento:** Los reportes estructurados y la documentación a través de *AsyncAPI* reducen la dependencia de conocimiento tácito y favorecen la colaboración entre equipos.
- **Habilitar integración continua:** El tiempo de configuración inferior a 10 minutos y la generación automatizada de reportes permiten integrar la herramienta en pipelines de CI/CD sin afectar significativamente los tiempos de construcción.

5.5. Trabajos futuros

La investigación realizada abrió múltiples líneas de trabajo que permitirían ampliar el alcance, mejorar las capacidades y fortalecer la validez externa de la herramienta desarrollada. A continuación se presentan los trabajos identificados:

5.5.1. Implementación de trazabilidad distribuida

La limitación más significativa identificada fue la validación evento por evento sin rastreo *end-to-end*. Una mejora valiosa consiste en implementar soporte para *correlation IDs* que permitan reconstruir trazas de transacciones distribuidas. Esto implica:

- **Enriquecimiento de metadatos:** Extraer y propagar *correlation IDs* desde los *headers*.
- **Almacenamiento de grafos:** Estructuras de datos para representar relaciones padre–hijo entre eventos.
- **Visualización de trazas:** Interfaces que muestren el flujo completo de una transacción.

Esta capacidad sería especialmente útil para validar patrones *SAGA* de larga duración.

5.5.2. Soporte para múltiples intermediarios de mensajes

La herramienta actual se especializa en RabbitMQ. Para ampliar su aplicabilidad sería necesario:

- **Abstracción de operaciones de mensajería:** Interfaces comunes para publicar, consumir y gestionar colas.
- **Adaptadores por plataforma:** Conectores específicos para Kafka, AWS SQS, Google Pub/Sub, Azure Service Bus.

Esto permitiría usar la herramienta en ecosistemas heterogéneos.

5.5.3. Instrumentación de métricas de rendimiento

Para evaluar la viabilidad de uso en entornos de alta demanda sería necesario incluir:

- Métricas de latencia de validación.
- Métricas de *throughput*.
- Monitoreo de CPU, memoria y ancho de banda.
- Dashboards de observabilidad en Grafana.

Estas métricas permitirían analizar el impacto real de la herramienta bajo carga.

5.5.4. Validación de patrones avanzados de mensajería

Los escenarios actuales se enfocan en patrones comunes. Extensiones adicionales incluirían:

- **Fanout exchanges.**
- **Headers exchanges.**
- **Delayed messages.**
- **Dead letter queues.**

Esto ampliaría la compatibilidad con configuraciones avanzadas de sistemas EDA.

5.5.5. Exportar reportes

La interfaz carece de opciones de exportación. Extensiones recomendadas:

- Reportes PDF completos con gráficos y tablas.
- Exportación JSON/XML para integración con sistemas externos.
- Generación automática de documentación (Markdown).

5.5.6. Aprendizaje automático para detección de anomalías.

Líneas prometedoras incluyen:

- Algoritmos no supervisados para detectar fallos.
- Modelos predictivos para anticipar fallos en flujos de eventos.

5.5.7. Validación en configuraciones de clúster

Como se fundamentó en la sección 2.1.2, el teorema CAP establece que los sistemas distribuidos deben elegir entre consistencia y disponibilidad ante particiones de red. RabbitMQ implementa diferentes modos de replicación (*quorum queues (CP), stream queues (AP) y mirrored queues (CP)*) que abordan de formas distintas este *trade-off*. Por lo anterior, un trabajo valioso consistiría en evaluar el comportamiento de la herramienta de pruebas en configuraciones de clúster:

- ***Colas quorum:*** Validar que la herramienta detecte correctamente eventos en escenarios donde la cola *quórum* se pierde temporalmente.
- ***Colas stream:*** Caracterizar el comportamiento de las colas espía ante réplicas con *lag* de replicación.
- **Particiones de red:** Evaluar cómo la herramienta maneja escenarios de *split-brain*.

5.6. Lecciones aprendidas

El proceso de desarrollo, implementación y validación de la herramienta proporcionó aprendizajes valiosos que trascienden los resultados técnicos específicos del proyecto. A continuación se documentan las lecciones más significativas, organizadas según su naturaleza técnica, metodológica o profesional.

5.6.1. Técnicas

- **Los estándares de la industria ayudan a la adopción:** El uso de AsyncAPI eliminó formatos propietarios y permitió reutilizar especificaciones existentes sin modificaciones, reduciendo la curva de aprendizaje.
- **El monitoreo no intrusivo habilita pruebas realistas:** Las *spy queues* permitieron capturar eventos sin modificar los servicios bajo prueba, mejorando validez de las evaluaciones.

5.6.2. Metodológicas

- **Design Thinking proporciona estructura efectiva para proyectos de ingeniería:** El enfoque empatizar–definir–idear–prototipar–testear permitió alinear el desarrollo técnico con necesidades reales y evitar sobreingeniería.
- **La triangulación de fuentes fortalece la validez de requisitos:** Entrevistas, encuestas, análisis documental y observación convergieron en necesidades consistentes.

5.6.3. Desarrollo asistido por IA

- **Los asistentes de IA amplifican productividad cuando se guían con contextos claros:** El uso de Cursor IDE y Claude Code permitió acelerar la generación de código repetitivo sin comprometer calidad.
- **La revisión humana sigue siendo indispensable.** A pesar de sugerencias correctas, algunos fragmentos generados requirieron refactorización o ajustes arquitectónicos.
- **La documentación de contexto es crítica para obtener resultados consistentes.** El archivo `CLAUDE.md` con información del dominio y arquitectura fue clave para obtener código coherente.

5.7. Reflexiones finales

El desarrollo de esta herramienta confirmó que la ingeniería de software aplicada a sistemas distribuidos requiere equilibrio entre rigor técnico y pragmatismo operativo. Asimismo, se evidenció que las brechas entre teoría académica y práctica profesional pueden reducirse significativamente mediante metodologías centradas en el usuario y validaciones inspiradas en escenarios reales. Finalmente, el proyecto demostró que documentación clara, automatización y adopción de estándares multiplican el valor de las herramientas técnicas mucho más allá de sus capacidades inmediatas.

Capítulo 6

Anexos

Anexo A. Guía de entrevista

1. ¿Describe cómo realizas pruebas en sistemas asincrónicos o basados en eventos?
2. ¿Qué herramientas utilizas normalmente para esas pruebas? ¿Funcionan bien?
3. ¿Cuál ha sido el mayor reto que has enfrentado probando flujos de eventos?
4. ¿Te ha costado rastrear eventos entre servicios? ¿Por qué crees que pasa esto?
5. ¿Cómo detectas errores intermitentes o condiciones de carrera en estos sistemas?
6. ¿Qué cambiarías de las herramientas actuales de pruebas si pudieras?
7. ¿Qué tan fácil o difícil te resulta simular escenarios reales durante las pruebas?
8. ¿Qué características o funciones debería tener una herramienta de pruebas “ideal” para ayudarte?
9. ¿Te ha pasado alguna prueba pasó en test, pero falló en producción? Describe la situación.
10. ¿Hay algo que no te pregunté pero crees importante mencionar sobre este tema?

Anexo B. Encuesta aplicada

1. **¿Cuál es tu rol principal en el equipo de desarrollo?** (Selecciona una opción)
   - QA Engineer
   - QA Lead
   - Backend Developer
   - Backend/Frontend Lead
   - Arquitecto de Software
   - DevOps / SRE
   - Product Manager
   - Otro: _____

2. **¿Tienes experiencia probando sistemas con RabbitMQ o arquitecturas orientadas a eventos?**
   - Sí
   - No

   Bifurcación si la respuesta anterior es si:

3. **¿Con qué frecuencia enfrentas problemas difíciles de depurar relacionados con eventos?** (Escala tipo Likert)
   - 1 = Nunca
   - 2 = Rara vez
   - 3 = Ocasionalmente
   - 4 = Frecuentemente
   - 5 = Todo el tiempo

4. **¿Qué tan confiables consideras tus herramientas actuales para detectar fallas asincrónicas?** (Escala tipo Likert)
   - 1 = Nada confiables
   - 2 = Poco confiables
   - 3 = Algo confiables
   - 4 = Confiables
   - 5 = Muy confiables

5. **¿Has tenido fallos en producción por problemas no detectados en pruebas asincrónicas?**
   - Sí
   - No
   - No estoy seguro
6. **Selecciona las dificultades que has enfrentado al probar sistemas basados en eventos:** (Puedes marcar más de una)
   - Eventos perdidos
   - Mensajes duplicados
   - Errores intermitentes
   - Condiciones de carrera
   - Falta de trazabilidad
   - Otras: _____
7. **¿Usas alguna de estas herramientas?** (Selecciona todas las que apliquen)
   - Postman
   - RabbitMQ Management Plugin
   - Pact / Contract Testing
   - Wireshark
   - Jest / Mocha / Chai
   - Otras: _____
8. **¿Qué características te gustaría que tuviera una nueva herramienta de pruebas para arquitecturas basadas en eventos?** (Respuesta abierta)

Anexo C. Prompt para análisis de entrevistas

```
Eres un asistente especializado en análisis cualitativo asistido por
   IA.
Tengo transcripciones de entrevistas semiestructuradas en formato de
   texto.
Quiero que realices un análisis sistemático siguiendo estos pasos
   metodológicos:

1. Preprocesamiento del texto:
   - Limpia las transcripciones eliminando muletillas, repeticiones y
      palabras vacías (stopwords).
   - Segmenta el texto en fragmentos significativos (por
      pregunta/respuesta o por párrafo).

2. Vectorización semántica:
   - Convierte cada fragmento en un vector semántico utilizando
      embeddings tipo Sentence-BERT (SBERT).

3. Agrupamiento (clústeres):
   - Aplica algoritmos de agrupamiento no supervisado.
   - Usa K-Means para un número predefinido de clústeres (ej. 5) y
      HDBSCAN para detección automática.

4. Interpretación de clústeres:
   - Para cada clúster, calcula los términos más representativos con
      TF-IDF.
   - Resume en una frase el tema central de cada clúster.

5. Resultados:
   - Entrega una tabla con el formato:
     Clúster | Tema identificado | Palabras clave (TF-IDF) | Citas
        representativas.
   - Incluye un párrafo de síntesis que explique cómo los hallazgos
      reflejan
     necesidades y limitaciones en pruebas de arquitecturas orientadas
        a eventos.

Tu respuesta debe tener un tono académico, con redacción clara y
   orientada a un informe de investigación.
```

Anexo D. Prompt para análisis de encuestas

```
Eres un asistente experto en análisis de datos y metodología de
   investigación.
Tienes un archivo CSV con resultados de una encuesta estructurada en
   las siguientes preguntas:

1. Preguntas categóricas nominales (P1, P2, P5): rol, experiencia y
   fallos en producción.
2. Preguntas tipo Likert (P3, P4): frecuencia de errores y
   confiabilidad percibida.
3. Preguntas de selección múltiple (P6, P7): dificultades encontradas
   y herramientas usadas.
4. Pregunta abierta (P8): características deseadas para una
   herramienta de pruebas.

Aplica las siguientes metodologías de análisis y genera una tabla
   resumen con los resultados y hallazgos principales.

### Metodologías a aplicar
- **Para P1, P2 y P5 (variables categóricas nominales):**
  - Calcular frecuencias absolutas y porcentajes por categoría.
  - Aplicar la prueba de independencia  ²  (Chi-cuadrado) para detectar
     asociaciones entre variables, por ejemplo:
      * Rol × Experiencia
      * Experiencia × Fallos en producción
  - Reportar los estadísticos  ², gl, p-valor y V de Cramér.

- **Para P3 y P4 (escalas Likert):**
  - Calcular medidas descriptivas: media, mediana, moda, desviación
     estándar (DE) y rango intercuartílico (IQR).
  - Calcular la correlación de Spearman  () para evaluar la relación
     entre frecuencia de errores y confiabilidad percibida.

- **Para P6 y P7 (preguntas de selección múltiple):**
  - Calcular el porcentaje de aparición de cada opción.
  - Generar una matriz de co-ocurrencia de respuestas binarias.
  - Identificar las tres combinaciones de respuestas más frecuentes.
  - Resumir la interpretación de las combinaciones más representativas.

- **Para P8 (pregunta abierta):**
  - Analizar las respuestas textuales con TF-IDF (Term -
     FrequencyInverse Document Frequency).
  - Aplicar agrupamiento K-Means para identificar temas recurrentes.
```

```
  - Presentar una tabla con: {Clúster, Tema principal, Palabras clave
      (mayor peso TF-IDF), Citas representativas}.
  - Redactar una breve síntesis que describa los ejes conceptuales
      emergentes.

### Formato de salida requerido
1. **Tabla 1 - Distribución y asociaciones entre variables categóricas
   (P1, P2, P5)**
   Incluir frecuencias, porcentajes, x², gl, p-valor y V de Cramér.
2. **Tabla 2 - Estadísticos descriptivos y correlación para variables
   Likert (P3, P4)**
   Incluir media, mediana, DE, IQR y  de Spearman.
3. **Tabla 3 - Análisis de selección múltiple (P6, P7)**
   Incluir % de aparición, principales co-ocurrencias y su
      interpretación.
4. **Tabla 4 - Clústeres temáticos de respuestas abiertas (P8)**
   Incluir tema principal, palabras clave TF-IDF y citas
      representativas.

Devuelve todas las tablas en formato Markdown o LaTeX y un resumen
   interpretativo por cada bloque de análisis.
```

Anexo E. Estructura general event-flow-core

```
event-flow-core/
   src/
      api/                                  # Capa de entrada - Bootstrap
         y configuración
         app.ts                             # Aplicación principal
         bootstrap/                         # Inyección de dependencias y
            servidor
             di.ts                          # Dependency Injection
             server.ts                      # Configuración del servidor
             socket.ts                      # Configuración de WebSockets
             factory/                       # Factories para instancias

      common/                               # Utilidades compartidas
         event/                             # Eventos base
             event.ts

      config/                               # Configuración de la
         aplicación

      internal/                             # Núcleo (DDD/Hexagonal)
         domain/                            # Capa de dominio
            services/                       # Servicios de dominio
               (validadores)

         usecase/                           # Casos de uso (lógica de
            aplicación)

         platform/                          # Capa de infraestructura
             http/                          # HTTP handlers y middleware
                handler/
                   model/                   # Modelos HTTP
                middleware/
                router/
                    v1/

             socket/                        # WebSocket handlers
                handler/
                    socket.ts
                    models/

             storage/                       # Persistencia
```

```
            mongodb/              # Repositorios MongoDB
               model/            # Modelos MongoDB
               pipelines/        # Agregaciones MongoDB
            rabbitmq/             # Cliente RabbitMQ
            r2/                   # Cliente Cloudflare R2

         proxy/                    # Proxies externos

         rest/                     # Cliente REST (RabbitMQ
            Management API)
            models/

         mocks/                    # Mocks para testing

   docs/                           # Documentación
      swagger/                     # Documentación Swagger
      evaluation_scenarios/        # Escenarios de evaluación
         (YAML)
      app-docs/                    # Documentación de la app

db/                                # Base de datos y scripts

dist/                              # Código compilado
   (TypeScript)
node_modules/                      # Dependencias
coverage/                          # Reportes de cobertura

.env                               # Variables de entorno
.gitignore
.prettierrc
jest.config.js                     # Configuración de Jest
jest.setup.ts                      # Setup de Jest
nodemon.json                       # Configuración nodemon
package.json                       # Dependencias y scripts
tsconfig.json                      # Configuración TypeScript
CLAUDE.md                          # Documentación del proyecto
README.md
```

Anexo F. Estructura general event-flow-app

```
event-flow-app/
   src/
     index.html
     main.ts
     styles.css
     theme.scss
     app/
        app.component.*
        app.config.ts
        app.routes.ts
        common/              # Módulos compartidos
        core/                # Servicios, interfaces, enums
        auth/                # Autenticación
        dashboard/           # Dashboard principal
          components/
             brokers/
             flows/
                executions/
                   execution-create/
                   execution-list/
                   test-case-results/
        settings/
        home/
        environment/
   angular.json
   package.json
   tsconfig.*.json
```

Anexo G. Detalle de escenarios de validación

EV1. Validación de cumplimiento de contratos

- **Necesidad relacionada:** Validación de contratos entre productores y consumidores.
- **Descripción del problema:** En arquitecturas orientadas a eventos, los productores y consumidores operan de forma independiente, lo que puede resultar en inconsistencias estructurales o semánticas en los mensajes intercambiados. Errores como campos faltantes y tipos de datos incorrectos generan fallos que pueden llegar a ser silenciosos y difíciles de diagnosticar.
- **Objetivo del escenario:** Demostrar que la herramienta detecta violaciones de contrato validando automáticamente los datos de los eventos con el esquema AsyncAPI.
- **Configuración inicial:**
  - Crear especificación AsyncAPI con esquema estricto de evento OrderCreated que incluye campos obligatorios: *orderId (string), customerId (string), amount (number), currency (string), timestamp (datetime)*
- **Ejecución de la prueba:**
  1. Caso válido: Se publica mensaje que cumple completamente el contrato
  2. Caso con campo faltante: Se publica mensaje sin el campo *currency*
  3. Caso con tipo incorrecto: Se publica mensaje con *amount* de tipo *string* en lugar de *number*
  4. Caso con campo adicional: Se publica mensaje con campo *extraField* no definido en el esquema
  5. Caso con valor inválido (enum): Se publica mensaje con *currency* en formato no permitido (ej: 'PES' en lugar de 'COL')
  6. Caso con formato inválido: Se publica mensaje con *timestamp* que no cumple el formato ISO 8601 (date-time)
  7. Caso con patrón inválido: Se publica mensaje con *orderId* que no cumple el patrón 'ORD-[0-9]5'
  8. Caso con rango inválido: Se publica mensaje con *amount* negativo (viola minimum: 0.01)
- **Resultados esperados:**
  - La herramienta registra el caso válido como exitoso

  - La herramienta detecta y registra cada violación de contrato, especificando:
    - Campo problemático
    - Tipo de violación (campo faltante, tipo incorrecto, valor fuera de dominio)
    - Valor esperado vs. valor recibido
  - Los resultados se almacenan en los caso de prueba asociados al evento

- **Métricas de éxito:**
  - Tasa de detección: 100 % de violaciones detectadas (7 de 7 casos inválidos)
  - Precisión de diagnóstico: Descripción exacta del campo, tipo de error y valor esperado vs recibido en cada caso
  - Trazabilidad: Cada resultado vinculado al mensaje específico con timestamp y metadata completa
  - Notificación en tiempo real: Resultados emitidos vía WebSocket inmediatamente después de la validación

EV2. Detección de eventos perdidos

- **Necesidad relacionada:** Trazabilidad de eventos, Monitoreo de métricas de calidad.
- **Descripción del problema:** En sistemas distribuidos con múltiples saltos, los mensajes pueden perderse debido a fallos de red, caídas de consumidores, configuraciones incorrectas de colas o problemas con el *acknowledge*. La falta de visibilidad de punta a punta dificulta identificar en qué punto del flujo ocurrió la pérdida del mensaje.
- **Objetivo del escenario:** Demostrar que la herramienta identifica eventos esperados que nunca fueron entregados o procesados dentro de un flujo.
- **Configuración inicial:**
  - Definir un flujo con tres eventos secuenciales: `OrderCreated` → `PaymentProcessed` → `OrderConfirmed`.
  - Preparar un productor que publique `OrderCreated` y un consumidor encargado de emitir `PaymentProcessed`.
- **Ejecución de la prueba:**
  1. **Caso exitoso:** El productor publica `OrderCreated`, el primer consumidor procesa y emite `PaymentProcessed`, y el segundo consumidor emite `OrderConfirmed`. El flujo se completa correctamente.

2. **Caso con pérdida en medio del flujo:** El productor publica `OrderCreated`, pero el consumidor falla antes de emitir `PaymentProcessed`.
3. **Caso con pérdida al final del flujo:** `OrderCreated` y `PaymentProcessed` se procesan correctamente, pero `OrderConfirmed` nunca se emite.

- **Resultados esperados:**
  - En el caso exitoso, la herramienta registra la secuencia completa de eventos.
  - En los casos con pérdida, la herramienta identifica:
    - El último evento recibido exitosamente.
    - El evento esperado que no llegó.
    - Un estado final `FAILED`.
- **Métricas de éxito:**
  - **Tasa de detección de pérdidas:** 100 % de los eventos faltantes identificados (3 de 3 casos).
  - **Precisión de localización:** Identificación exacta del evento faltante dentro de la secuencia monitoreada.
  - **Completitud de contexto:** Registro de todos los eventos previos exitosos para facilitar el análisis de la causa raíz.

EV3. Detección de mensajes duplicados

- **Necesidad relacionada:** Monitoreo de métricas de calidad, Trazabilidad de eventos.
- **Descripción del problema:** Los reintentos automáticos, problemas de idempotencia y configuraciones incorrectas de consumidores pueden generar múltiples procesamientos del mismo mensaje. Esta duplicación puede provocar inconsistencias en el estado del sistema, tales como cobros dobles o desactualización del inventario.
- **Objetivo del escenario:** Demostrar que la herramienta detecta cuando un mismo evento lógico es procesado más de una vez.
- **Configuración inicial:**
  - Definir el evento `PaymentProcessed` con un identificador único `paymentId`.
  - Configurar un consumidor encargado de procesar pagos.
  - Preparar un inyector de fallos que simula duplicación de mensajes.
- **Ejecución de la prueba:**

1. **Caso sin duplicación:** El mensaje con `paymentId: "PAY-001"` se publica y procesa una sola vez.
2. **Caso con duplicación inmediata:** El mensaje con `paymentId: "PAY-002"` se publica dos veces en un tiempo menor a 1 segundo.
3. **Caso con duplicación retardada:** El mensaje con `paymentId: "PAY-003"` se procesa, y luego se reenvía después de 5 segundos.
4. **Caso con múltiples duplicaciones:** El mensaje con `paymentId: "PAY-004"` se procesa cinco veces.

- **Resultados esperados:**
  - La herramienta identifica los mensajes duplicados mediante:
    - Comparación del identificador único dentro del *payload*.
    - Registro de *timestamps* de cada ocurrencia.
    - Conteo de procesamientos por identificador.
  - Para cada duplicación detectada, la herramienta registra:
    - El identificador del mensaje duplicado.
    - El número total de ocurrencias.
    - El intervalo temporal entre duplicaciones.
  - El dashboard muestra la tasa de duplicación por flujo.
- **Métricas de éxito:**
  - **Tasa de detección:** 100 % de duplicaciones identificadas (3 de 3 casos).
  - **Granularidad temporal:** Registro de *timestamps* con precisión de milisegundos.
  - **Escalabilidad:** Detección efectiva hasta 10 duplicaciones del mismo mensaje.

EV4. Validación de orden de eventos

- **Necesidad relacionada:** Trazabilidad de eventos, Simulación de escenarios reales.
- **Descripción del problema:** Muchos flujos de negocio requieren que los eventos se procesen en un orden específico, por ejemplo, `OrderCreated` debe preceder a `OrderShipped`. Sin embargo, condiciones propias de sistemas distribuidos (como latencia de red, paralelismo o múltiples consumidores) pueden ocasionar que los eventos se procesen fuera del orden esperado, generando estados inválidos en el sistema.

- **Objetivo del escenario:** Demostrar que la herramienta valida el orden esperado de los eventos en flujos secuenciales.
- **Configuración inicial:**
  - Definir un flujo con orden estricto: `OrderCreated` (1) → `PaymentProcessed` (2) → `OrderShipped` (3) → `OrderDelivered` (4).
  - Configurar la ejecución con `orderMode: 'strict'`.
  - Preparar productores independientes para cada tipo de evento.
- **Ejecución de la prueba:**
  1. **Caso con orden correcto:** Los eventos se publican y procesan en la secuencia 1→2→3→4.
  2. **Caso con inversión parcial:** Los eventos llegan en orden 1→3→2→4 (es decir, `OrderShipped` antes de `PaymentProcessed`).
  3. **Caso con evento adelantado:** `OrderDelivered` arriba antes de cualquier otro evento.
- **Resultados esperados:**
  - **Caso 1:** Ejecución `SUCCEEDED`; todos los eventos marcados como `PASSED` con orden correcto.
  - **Caso 2:** Ejecución `FAILED`; la herramienta identifica:
    - Evento `OrderShipped` recibido fuera de su posición esperada.
    - Estado del evento: `FAILED`
  - **Caso 3:** Ejecución `FAILED` inmediatamente al recibir `OrderDelivered` sin eventos previos.
- **Métricas de éxito:**
  - **Detección de violaciones de orden:** 100 % de casos fuera de orden identificados (2 de 2).
  - **Precisión de posición:** Identificación exacta del evento problemático y su posición esperada vs. posición real.
  - **Claridad de reporte:** Mensajes descriptivos que indican el orden esperado y el recibido.

EV5. Validación de configuración de infraestructura

- **Necesidad relacionada:** Simulación de escenarios reales, Estandarización del proceso de prueba.
- **Descripción del problema:** Errores de configuración en RabbitMQ como *exchanges* sin colas, colas sin *consumers* o *bindings* incorrectos pueden provocar pérdida silenciosa de mensajes. Estos problemas suelen descubrirse tardíamente, incluso después de desplegar en producción.
- **Objetivo del escenario:** Demostrar que la herramienta valida la configuración de infraestructura antes de ejecutar las pruebas.
- **Configuración inicial:**
  - La especificación AsyncAPI define eventos junto con sus *exchanges*, *routing keys* y colas dentro del flujo.
- **Ejecución de la prueba:**
  1. **Configuración válida:** El *exchange* existe, tiene una cola enlazada con la *routing key* correcta.
  2. **Exchange sin colas:** El *exchange* `orders.events` existe pero no tiene colas vinculadas.
  3. **Routing key incorrecto:** La cola está enlazada al *exchange* pero usando una *routing key* distinta a la especificada.
- **Resultados esperados:**
  - **Para configuración válida:**
    - El reporte muestra: `Exchange`, `Binding`.
  - **Para configuraciones inválidas:**
    - El reporte indica específicamente el problema detectado.
    - El estado de la ejecución se marca como `FAILED` con un mensaje descriptivo.
- **Métricas de éxito:**
  - **Cobertura de validación:** Verificación de: *exchanges* y *bindings*.
  - **Detección preventiva:** 100 % de configuraciones inválidas detectadas antes de la ejecución (2 de 2 casos).
  - **Tiempo de validación:** Menos de 5 segundos para validar la configuración completa.

- **Claridad de diagnóstico:** Mensajes específicos indicando el componente afectado y el error encontrado.

EV6. Detección de eventos huérfanos

- **Necesidad relacionada:** Validación de contratos, Detección de inconsistencias en la infraestructura.
- **Descripción del problema:** En sistemas basados en eventos, pueden llegar mensajes con routing keys que no están definidos en la documentación AsyncAPI del sistema. Estos eventos 'huérfanos' representan mensajes no esperados que pueden indicar:
  - Servicios legacy enviando eventos obsoletos.
  - Errores de configuración en productores.
  - Documentación desactualizada
  - Posibles intentos de inyección de datos

  Detectar eventos huérfanos es fundamental para mantener la coherencia entre la documentación y el comportamiento real del sistema.
- **Objetivo del escenario:** Demostrar que la herramienta detecta eventos que llegan al broker pero no están definidos en la especificación AsyncAPI, identificando discrepancias entre la infraestructura real y la documentada.
- **Configuración inicial:**
  - Definir una especificación AsyncAPI con eventos específicos, por ejemplo: `OrderCreated` (RK: `order.created`), `PaymentProcessed` (RK: `payment.processed`).
- **Ejecución de la prueba:**
  1. **Evento documentado:** Llega `OrderCreated` con RK `order.created` definido en AsyncAPI.
  2. **Evento huérfano:** Llega un mensaje con RK `order.cancelled`, que NO está en la especificación AsyncAPI.
- **Resultados esperados:**
  - **Caso 1:** Los eventos `order.created, payment.processed` NO son reportados como huérfanos.

- **Caso 2:** El test case `orphan_event` se marca como `FAILED`, reportando RK `order.cancelled` como evento sin infraestructura.
- El dashboard muestra los detalles completos del evento huérfano: routing key, exchange y contenido del mensaje.

▪ **Métricas de éxito:**

- **Detección de huérfanos:** 100 % de eventos con routing keys no documentados son identificados (2 de 2 casos).
- **Cobertura de monitoreo:** Todos los exchanges definidos en AsyncAPI son monitoreados con *wildcard routing key*.
- **Especificidad de reporte:** Cada evento huérfano incluye routing key exacto, exchange y payload completo para facilitar la depuración.
- **Sin falsos positivos:** Eventos documentados NO son marcados como huérfanos.

Bibliografía

Alkharusi, H. (2022). A descriptive analysis and interpretation of data from likert scales in educational and psychological research. *Indian Journal of Psychology and Education*, 12(2):13–16.

Amazon Web Services (2024). @aws-sdk/client-s3 — aws sdk for javascript v3. https://docs.aws.amazon.com/AWSJavaScriptSDK/v3/latest/clients/client-s3/. Consultado en 2025.

Anthropic (2025). Claude code – agentic coding tool by anthropic. https://code.claude.com/docs/en/overview. Consultado en 2025.

Apache Software Foundation (2025). *Apache JMeter User Manual: Getting Started*. Accedido el 31 de agosto de 2025.

AsyncAPI Initiative (2024). @asyncapi/parser — asyncapi parser library. https://www.asyncapi.com/docs/tools/generator/parser#main-content. Consultado en 2025.

Auth0 Inc. (2024). jsonwebtoken — implementation of json web tokens. https://github.com/auth0/node-jsonwebtoken. Consultado en 2025.

Babich, N. (2019). The art of the user interview. https://medium.springboard.com/the-art-of-the-user-interview-cf40d1ca62e8. Medium.

Barke, S., James, M. B., and Polikarpova, N. (2022). Grounded copilot: How programmers interact with code-generating models. In *Proceedings of the 44th International Conference on Software Engineering (ICSE 2022)*, page 1460–1472. IEEE.

bcrypt Project Contributors (2024). bcrypt — a library to hash passwords. https://www.npmjs.com/package/bcrypt. Consultado en 2025.

Boehm, B. and Basili, V. R. (2001). Software defect reduction top 10 list. *IEEE Computer*, 34(1):135–137.

Brandt, C. (2025). Towards refined code coverage: A new predictive problem in software testing. In *Proceedings of the 2025 IEEE/ACM International Conference on Software Testing, Verification and Validation (ICST)*, New York, NY. IEEE.

Brewer, E. A. (2012). CAP twelve years later: How the “rules” have changed. *Computer*, 45(2):23–29.

Chen, L. and Liu, L. (2020). Methods to analyze likert-type data in educational technology research. *Journal of Educational Technology Development and Exchange (JETDE)*, 13(2):1–19.

Consortium for Information & Software Quality (CISQ) (2022). The cost of poor software quality in the us: A 2022 report. https://www.it-cisq.org/wp-content/uploads/sites/6/2022/11/CPSQ-Report-Nov-22-2.pdf. Recuperado el 2 de abril de 2025.

Cursor Inc. (2025). Cursor – the best way to code with ai. https://cursor.com/docs. Consultado en 2025.

Datadog (2022). How to monitor event-driven architectures. https://www.datadoghq.com/blog/monitor-event-driven-architectures/. Recuperado el 2 de abril de 2025.

dotenv Project (2024). dotenv — loads environment variables from .env files. https://www.npmjs.com/package/dotenv. Consultado en 2025.

Express.js Foundation (2024a). cors — node.js middleware for enabling cross-origin resource sharing. https://www.npmjs.com/package/cors. Consultado en 2025.

Express.js Foundation (2024b). Express.js — fast, unopinionated, minimalist web framework for node.js. https://expressjs.com/. Consultado en 2025.

Express.js Foundation (2024c). Multer — node.js middleware for handling multipart/form-data. https://github.com/expressjs/multer. Consultado en 2025.

Gilbert, S. and Lynch, N. (2002). Brewer's conjecture and the feasibility of consistent, available, partition-tolerant web services. *ACM SIGACT News*, 33(2):51–59.

González-Barahona, J. M. and Robles, G. (2012). On the reproducibility of empirical software engineering studies based on data retrieved from development repositories. *Empirical Software Engineering*, 17(1):75–89.

Google Cloud and DORA Research Program (2025). 2025 state of ai-assisted software development report. Technical report, DevOps Research and Assessment (DORA). Accessed: November 2025.

Google LLC and Angular Material Team (2024). @angular/material — angular material design components. https://material.angular.io/. Consultado en 2025.

Google LLC and Angular Team (2024). @angular/core — angular framework core library. https://angular.dev/guide. Consultado en 2025.

Headlee, C. (2015). 10 ways to have a better conversation. https://www.ted.com/talks/celeste$_h$eadlee$_1$0$_w$ays$_t$o$_h$ave$_a$better$_c$onversation?subtitle = es.TEDTalk.

Herbold, S. and Harms, P. (2013). Autoquest – automated quality engineering of event-driven software. In *Proceedings of the 2013 IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, pages 134–139. IEEE.

Isaac Z. (2024). rimraf — the unix rm -rf command for node.js. https://www.npmjs.com/package/rimraf. Consultado en 2025.

Klemmer, J. H., Horstmann, S. A., Patnaik, N., Ludden, C., et al. (2024). Using ai assistants in software development: A qualitative study on security practices and concerns. *Empirical Software Engineering*, 29(4):1–30.

Kleppmann, M. (2017). *Designing Data-Intensive Applications: The Big Ideas Behind Reliable, Scalable, and Maintainable Systems*. O'Reilly Media, Sebastopol, CA.

Koziol, N. A. and Bilder, C. R. (2014). Mrcv: A package for analyzing categorical variables with multiple response options. *The R Journal*, 6(2):144–150.

Lee, Y. (2009). Event-driven soa test framework based on bpa-simulation. In *Proceedings of the 2009 First International Conference on Networked Digital Technologies (NDT)*, pages 189–194. IEEE.

Leifer, L., Link, P., and Lewrick, M. (2020). *The Design Thinking Toolbox: A Guide to Mastering the Most Popular and Valuable Innovation Methods*. Wiley, Hoboken, NJ.

Malzer, C. and Baum, M. (2019). A hybrid approach to hierarchical density-based cluster selection. *arXiv preprint arXiv:1911.02282*.

Martin, R. C. (2008). *Clean Code: A Handbook of Agile Software Craftsmanship*. Prentice Hall, Upper Saddle River, NJ.

McHugh, M. L. (2013). The chi-square test of independence. *Biochemia Medica*, 23(2):143–149.

Meta Open Source (2024). Jest — delightful javascript testing framework with a focus on simplicity. https://jestjs.io/. Consultado en 2025.

Microsoft (2024). Typescript handbook: Documentation and language overview. https://www.typescriptlang.org/docs/handbook/typescript-in-5-minutes.html. Consultado en 2025.

MongoDB Inc. (2024). Mongodb manual — document database for modern applications. https://www.mongodb.com/docs/. Consultado en 2025.

Nodeca (2024). js-yaml — yaml parser and serializer for javascript. https://www.npmjs.com/package/js-yaml. Consultado en 2025.

Okonetchnikov, Andrey (2024). lint-staged — run linters on git staged files. https://github.com/okonet/lint-staged. Consultado en 2025.

Ongaro, D. and Ousterhout, J. (2014). In search of an understandable consensus algorithm. In *Proceedings of the 2014 USENIX Annual Technical Conference (USENIX ATC)*, pages 305–319. USENIX Association.

Postman (2025). *Postman Learning Center*. Accedido el 31 de agosto de 2025.

Prettier Core Team (2024). Prettier — opinionated code formatter. https://prettier.io/. Consultado en 2025.

RabbitMQ (2024a). Clustering guide. https://www.rabbitmq.com/docs/clustering. Accessed: December 2025.

RabbitMQ (2024b). Quorum queues. https://www.rabbitmq.com/docs/quorum-queues. Accessed: December 2025.

RabbitMQ (2024c). Streams. https://www.rabbitmq.com/docs/streams. Accessed: December 2025.

Rahmatulloh, A., Nugraha, F., Gunawan, R., Darmawan, I., Haerani, E., and Rizal, R. (2024). Event-driven architecture (eda) vs api-driven architecture (ada): Which performs better in microservices? In *2024 International Conference on Artificial Intelligence, Blockchain, Cloud Computing, and Data Analytics (ICoABCD)*, pages 31–36.

ReactiveX Contributors (2024). Rxjs — reactive extensions library for javascript. https://rxjs.dev/. Consultado en 2025.

Reimers, N. and Gurevych, I. (2019). Sentence-bert: Sentence embeddings using siamese bert-networks. In *Proceedings of the 2019 Conference on Empirical Methods in Natural Language Processing (EMNLP)*. Association for Computational Linguistics.

Remy Sharp (2024). Nodemon — automatically restart node.js apps when file changes are detected. https://nodemon.io/. Consultado en 2025.

Rivera, D. (2024). @edaniel30/loki-logger — biblioteca de integración con grafana loki. https://github.com/edaniel30/loki-logger/pkgs/npm/loki-logger. Biblioteca desarrollada por el autor para la centralización de logs y métricas en Grafana Loki. Consultado en 2025.

Sampath, M. and Vignesh, S. (2024). Text clustering for topic identification: A tf-idf and k-means approach applied to the 20 newsgroups dataset. *Reva University Preprints*.

Silva, M. V. S., dos Santos, L. F. C., Soares, M. S., and Rocha, F. G. (2025). Guidelines for the application of event driven architecture in micro services with high volume of data. In *Proceedings of the 27th International Conference on Enterprise Information Systems (ICEIS 2025) - Volume 2*, pages 859–866. SCITEPRESS.

Socket.IO Team (2024). Socket.io — bidirectional and low-latency communication for every platform. https://socket.io/docs/. Consultado en 2025.

SquareMo (2024). amqplib — amqp 0-9-1 library and client for node.js. https://www.npmjs.com/package/amqplib. Consultado en 2025.

Swagger OpenAPI Initiative (2024). Swagger jsdoc — api documentation generator for node.js. https://www.npmjs.com/package/swagger-jsdoc. Consultado en 2025.

SweetAlert2 Team (2024). Sweetalert2 — beautiful, responsive, customizable javascript alerts. https://sweetalert2.github.io/. Consultado en 2025.

TheirStack (s.f.). Rabbitmq: Tecnología y usos en sistemas distribuidos. Recuperado de https://theirstack.com/es/technology/rabbitmq.

Treude, C. (2023). How developers interact with ai: A taxonomy of human-ai collaboration in software engineering. In *Proceedings of the 2023 IEEE/ACM 45th International Conference on Software Engineering (ICSE)*, pages 1–13. IEEE.

typestack (2024). class-validator — validation library for typescript and javascript. https://github.com/typestack/class-validator. Consultado en 2025.

TypeStrong (2024). ts-node — typescript execution environment for node.js. https://typestrong.org/ts-node/. Consultado en 2025.

Typicode (2024). Husky — modern native git hooks made easy. https://typicode.github.io/husky/. Consultado en 2025.

Vernon, V. (2016). *Domain-Driven Design Destilado*. Addison-Wesley Professional, Boston, MA.

Wu, C.-F., Ma, S.-P., Shau, A.-C., and Yeh, H.-W. (2022). Testing for event-driven microservices based on consumer-driven contracts and state models. In *Proceedings of the 29th Asia-Pacific Software Engineering Conference (APSEC)*, pages 467–471. IEEE.