Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Experience Report: Mimick

Tesis de Licenciatura en Ciencias de la Computación

Franco A. Castellacci

Director: Diego Garbervetsky
Buenos Aires, 2025

EXPERIENCE REPORT: MIMICK

En el desarrollo de software orientado a servicios, particularmente en el sector bancario y financiero (BFSI), las arquitecturas distribuidas generan dependencias críticas de servicios externos que se manifiestan en múltiples obstáculos operacionales. Estas dependencias impactan directamente los ciclos de desarrollo mediante la dificultad para generar datos de prueba realistas, tiempos de respuesta inconsistentes que dificultan el diagnóstico de problemas, y la disponibilidad periódica de servicios durante fases críticas de integración y testing. A pesar de la existencia de diversas herramientas de Service Virtualization en el mercado, las soluciones disponibles presentan una limitación fundamental: la ausencia de mecanismos de aislamiento por sesión de usuario, lo cual genera conflictos entre equipos de desarrollo que necesitan simular diferentes estados del mismo servicio simultáneamente. Este escenario resulta en trabajo duplicado, estados inconsistentes durante las pruebas, y una coordinación explícita entre desarrolladores que ralentiza el progreso colectivo. La problemática se intensifica en entornos donde múltiples equipos de desarrollo y Quality Assurance (QA) comparten infraestructura de testing, donde la configuración global de mocks genera interferencias inevitables que comprometen tanto la productividad como la calidad del software entregado. Este trabajo documenta el diseño, implementación y evaluación de Mimick, una plataforma de simulación de servicios que incorpora tres componentes diferenciadores: aislamiento por sesión mediante identificadores únicos que permiten configuraciones independientes por usuario, un proxy híbrido configurable para redirección granular de tráfico, y generación automática de especificaciones desde tráfico HTTP capturado.

Adicionalmente, este reporte de experiencia examina el proceso que originó la herramienta, incluyendo la exploración inicial de IA Generativa para la resolución automática de respuestas y el posterior cambio hacia un enfoque determinístico centrado en captura directa y reproducción fiel de tráfico, decisión que resultó más efectiva y alineada con las necesidades de predictibilidad que caracterizan las herramientas de desarrollo.

Palabras clave: *Service Virtualization, Mocking, Proxy HTTP, Testing, Aislamiento por sesión, Desarrollo colaborativo, APIs REST, Sector BFSI.*

Agradecimientos

A mi familia, que siempre me apoyó y me dio aliento.
A Male, que mejora cada aspecto de mi vida.
A Gabriel, Victor y Gonzalo, sin ellos no hubiese llegado hasta acá
A mis amigos y amigas, que hacen que todo sea mejor.
A mis compañeros de la facultad.
A mis docentes, a la UBA y a la universidad pública.
A Diego Garbervetsky, que logró que pase algo que ya había dado por imposible.
A Fede y a Nico por insistir.
A todos los que siempre estuvieron y a los que ya no están.

Índice general

1. Introducción **8**
1.1 Contexto y problemática 8
1.2 Contribuciones de la tesis 9
1.3 Estructura de la tesis 9
2. Definiciones **9**
2.1 Desarrollo de software y dependencias externas 9
2.2 Simulación de servicios (Service Virtualization) 10
2.2.1 Características principales de la simulación de servicios 10
2.2.2 Beneficios de la simulación de servicios 11
2.3 Análisis de herramientas existentes 12
2.3.1 WireMock 12
2.3.2 Beeceptor 12
2.3.3 Postman 13
2.3.4 Análisis comparativo 13
3. Motivación y análisis del problema 14
3.1 Identificación del problema 15
3.1.1 Dependencias de servicios de terceros en el desarrollo de software 15
3.3 Análisis de mercado y justificación 16
3.4 Objetivos del proyecto y criterios de éxito 16
4. Propuesta de solución 17
4.1 Visión general de Mimick 18
4.2 Componentes diferenciadores 18
4.2.1 Proxy 19
4.2.2 Manejo de sesión de usuarios 19
4.2.3 Grabado de Tráfico 20
5. Arquitectura del sistema 21
5.1 Diseño arquitectónico general 22
5.2 Componentes principales: Proxy, APIs y interfaz web 24
5.2.1 Proxy basado en NGINX y OpenResty 24
5.2.2 Proxy API 25
5.2.3 API Mimick 26
5.2.5 Interfaz Web 29
5.4 Análisis de riesgos 29
6. Desarrollo e implementación **31**
6.1 Fases de Desarrollo: PoC, MVP y Refinamiento Continuo 32
6.1.1 Proof of Concept: Validación del modelo operacional 32
6.1.2 Minimum Viable Product: Primera versión funcional 32
6.1.3 Iteraciones posteriores: Refinamiento continuo 33

6.1.4 Consideraciones sobre el enfoque inicial con inteligencia artificial 34
6.2 Desafíos técnicos y soluciones implementadas 35
6.2.1 Modelado de estado para sobreescritura de respuestas 35
6.2.2 Ruteo dinámico en el Proxy 36
6.2.3 Matching complejo de requests 37
7. Despliegue y evaluación 38
7.1 Puesta en producción 38
7.2 Testing y validación 39
7.2.1 Testing facilitado por Mimick 39
7.2.2 Testing del sistema Mimick 40
7.3 Metodología de evaluación 41
7.4 Resultados obtenidos 42
7.4.1 Métricas cuantitativas 42
7.4.2 Feedback cualitativo 42
7.5 Comparación con herramientas existentes 43
7.5.1 Mimick versus WireMock 44
7.5.2 Mimick versus Beeceptor 44
7.5.3 Mimick versus Postman Mock Server 44
7.5.4 Limitaciones de Mimick 45
8. Lecciones aprendidas 46
8.1 Aprendizajes técnicos 47
8.2 Reflexiones sobre la transición del enfoque basado en IA hacia un enfoque determinístico 48
8.2.1 Motivación inicial del enfoque basado en IA 48
8.2.2 Exploración técnica y modelos evaluados 48
8.2.3 Problemas técnicos identificados 49
8.2.4 Problemas funcionales y predictibilidad 49
8.2.5 Proceso de decisión y momento del cambio de enfoque 50
8.2.6 Lecciones generalizables sobre aplicación de IA 51
9. Conclusiones y trabajo futuro 51
9.1 Síntesis de resultados 51
9.2 Cumplimiento de objetivos 52
9.3 Limitaciones y trabajo futuro 53
9.4 Reflexiones finales 54
Bibliografia 55

1. Introducción

La simulación de servicios ha emergido como una disciplina fundamental en el desarrollo de software, especialmente en arquitecturas distribuidas donde la interdependencia entre componentes puede convertirse en un cuello de botella crítico para la productividad de los equipos de desarrollo. La capacidad de replicar el comportamiento de servicios externos de manera controlada y predecible no sólo acelera los ciclos de desarrollo, sino que también mejora la calidad y confiabilidad del software entregado [1].

1.1 Contexto y problemática

El desarrollo de software orientado a servicios se caracteriza por la creciente complejidad de los sistemas distribuidos y la dependencia de múltiples servicios externos para completar funcionalidades críticas de negocio [2]. En entornos de desarrollo a gran escala, esta dependencia se ha convertido en un factor limitante que genera múltiples obstáculos operacionales que impactan directamente en la productividad de los equipos de desarrollo y Quality Assurance (QA).

Los problemas más significativos incluyen la dificultad para generar datos de prueba realistas y casos borde específicos, tiempos de respuesta lentos que impactan en los ciclos de desarrollo, y las caídas de servicios críticos durante las fases de integración y testing. Estas limitaciones se traducen en retrasos en los ciclos de entrega, incremento en la tasa de errores en producción, y una reducción general en la eficiencia del proceso de desarrollo de software [3].

Para mitigar estos problemas, la industria ha desarrollado diversas soluciones de simulación de servicios, también conocidas como *service virtualization*. Herramientas como WireMock [4], Postman Mock Server [5] y Beeceptor [6] ofrecen capacidades básicas de simulación de APIs REST [7], permitiendo a los equipos de desarrollo trabajar de manera más independiente de los servicios externos. Sin embargo, un análisis exhaustivo de estas soluciones revela limitaciones críticas para entornos de desarrollo colaborativo.

Las principales limitaciones identificadas en las herramientas existentes se centran en la configuración global compartida que genera conflictos cuando múltiples desarrolladores necesitan simular diferentes estados del mismo servicio. La ausencia de gestión de sesiones por usuario individual imposibilita escenarios de prueba personalizados, mientras que la complejidad en la configuración de escenarios dinámicos que requieren estado persistente y evolución temporal dificulta el testing exhaustivo. Adicionalmente, la capacidad limitada de autogeneración de especificaciones a partir de tráfico real requiere configuración manual extensiva que ralentiza la adopción y uso de estas herramientas.

Las soluciones *enterprise* disponibles en el mercado requieren licenciamiento costoso y configuraciones complejas que las hacen poco viables para equipos de desarrollo ágiles con recursos limitados. Esta situación resulta en que muchos equipos continúen dependiendo de servicios externos reales durante el desarrollo, perpetuando los problemas operacionales mencionados.

El impacto de estas limitaciones se manifiesta en bloqueos entre equipos de desarrollo, trabajo duplicado en la configuración de entornos de prueba, y la imposibilidad de mantener estados coherentes durante ciclos de testing extensos. Estos factores no solo afectan la velocidad de desarrollo, sino que también comprometen la calidad del software entregado al dificultar la implementación de pruebas exhaustivas y reproducibles.

1.2 Contribuciones de la tesis

Esta tesis presenta varias contribuciones principales al campo de la simulación de servicios y el desarrollo de software. En primer lugar, se documenta el diseño e implementación de Mimick, una herramienta integral para simulación de servicios que incorpora capacidades avanzadas de proxy, manejo de sesiones y grabado de tráfico, desarrollada específicamente para abordar las limitaciones identificadas en las herramientas existentes. Complementariamente, se realiza una evaluación comparativa exhaustiva que incluye un análisis detallado de herramientas existentes en el mercado como WireMock, Beeceptor y Postman, identificando sus fortalezas y limitaciones.

También se analizan las lecciones surgidas del cambio de dirección tecnológica, en particular la transición desde un enfoque basado en IA hacia una solución determinística. Este análisis permite identificar los contextos en los que la IA aporta valor en herramientas de desarrollo, especialmente en dominios donde la predictibilidad es esencial.

1.3 Estructura de la tesis

Esta tesis está organizada en los siguientes capítulos. El Capítulo 2 presenta las definiciones preliminares necesarias para comprender los conceptos fundamentales de simulación de servicios, desarrollo de software con dependencias externas, y las herramientas existentes en el mercado. El Capítulo 3 profundiza en la motivación específica del proyecto y proporciona un análisis detallado del problema, incluyendo el impacto en equipos de desarrollo y el análisis de mercado que justifica la propuesta. El Capítulo 4 describe la propuesta de solución, presentando la visión general de Mimick y sus componentes diferenciadores. El Capítulo 5 detalla la arquitectura del sistema, incluyendo el diseño arquitectónico general y el análisis de riesgos asociados. El Capítulo 6 documenta el proceso de desarrollo e implementación, desde las fases iniciales hasta las iteraciones finales. El Capítulo 7 presenta el despliegue y evaluación de la herramienta, incluyendo la metodología de evaluación y los resultados obtenidos. El Capítulo 8 reflexiona sobre las lecciones aprendidas durante el desarrollo del proyecto. Finalmente, el Capítulo 9 presenta las conclusiones, síntesis de resultados y propuestas para trabajo futuro.

2. Definiciones

2.1 Desarrollo de software y dependencias externas

El desarrollo de software orientado a servicios se caracteriza por la construcción de aplicaciones complejas que rara vez operan de forma aislada. En lugar de desarrollar todos los componentes desde cero, los equipos de desarrollo aprovechan servicios externos, APIs de terceros, y microservicios distribuidos para acelerar el proceso de creación de software y aprovechar funcionalidades especializadas.

Las dependencias externas en el contexto del desarrollo de software se refieren a todos aquellos servicios, APIs, bases de datos, o componentes que no están bajo el control directo del equipo de desarrollo pero que son necesarios para el correcto funcionamiento de la aplicación. Estas dependencias pueden incluir servicios de autenticación, sistemas de pago, APIs de geolocalización, servicios de

notificaciones, bases de datos compartidas, y cualquier otro servicio que proporcione funcionalidad crítica para la aplicación.

Sin embargo, esta dependencia de servicios externos introduce múltiples desafíos operacionales que impactan directamente en la eficiencia del proceso de desarrollo. La disponibilidad inconsistente de estos servicios constituye un problema recurrente, dado que pueden experimentar caídas no planificadas, mantenimientos programados, o problemas de conectividad que bloquean completamente el progreso del desarrollo local. Cuando un servicio crítico no está disponible, los desarrolladores no pueden probar funcionalidades completas ni validar integraciones.

Los tiempos de respuesta variables representan otro desafío significativo, donde la latencia de red y la carga en los servicios externos puede resultar en tiempos de respuesta lentos que afectan la productividad durante el desarrollo y testing. Esto es particularmente problemático cuando se ejecutan pruebas automatizadas que requieren múltiples llamadas a servicios externos. Además, esta variabilidad dificulta significativamente el análisis de performance del sistema de forma integral, dado que resulta imposible distinguir entre problemas de performance inherentes a la aplicación versus latencias introducidas por servicios externos.

La generación de datos de prueba presenta limitaciones adicionales, dado que los servicios externos frecuentemente tienen restricciones en cuanto a la creación de datos de prueba, especialmente para casos borde o escenarios de error. Esto limita la capacidad de los equipos de QA para validar comportamientos específicos y puede resultar en errores no detectados que llegan a producción [8]. En entornos colaborativos donde múltiples equipos de desarrollo comparten los mismos servicios externos para testing, pueden surgir conflictos de estado donde las acciones de un equipo afectan los datos o configuraciones que otros equipos están utilizando [9]. Esta interferencia entre equipos genera bloqueos y requiere coordinación explícita que ralentiza el proceso de desarrollo.

Los costos asociados con el uso de servicios externos durante desarrollo y testing constituyen una consideración adicional. Muchos servicios externos cobran por uso, lo que puede resultar en costos significativos especialmente cuando se ejecutan pruebas automatizadas repetitivas [10]. Estos desafíos se intensifican en entornos de desarrollo ágil donde los ciclos de iteración son rápidos y la capacidad de realizar pruebas frecuentes es fundamental para mantener la calidad del software [11].

2.2 Simulación de servicios (Service Virtualization)

La simulación de servicios, también conocida como Service Virtualization [1]??, es una técnica de testing que consiste en crear representaciones virtuales de servicios externos para ser utilizadas durante las fases de desarrollo, testing e integración. Esta práctica permite a los equipos de desarrollo trabajar de manera independiente sin depender de la disponibilidad real de los servicios externos.

El concepto de Service Virtualization surgió como respuesta a los desafíos inherentes del desarrollo de software en arquitecturas distribuidas y orientadas a servicios. A medida que las aplicaciones se volvieron más complejas y dependientes de múltiples servicios externos, se hizo evidente la necesidad de herramientas que permitieran simular estos servicios de manera controlada y predecible [12].

2.2.1 Características principales de la simulación de servicios

Los servicios virtualizados replican el comportamiento de los servicios reales de manera comprehensiva, incluyendo las respuestas esperadas, códigos de estado HTTP, headers, y estructura de datos. Esta emulación permite que las aplicaciones interactúen con los servicios simulados de la misma

manera que lo harían con los servicios reales, garantizando que el código de integración funcione correctamente cuando eventualmente se conecte a los servicios productivos.

La disponibilidad garantizada constituye una característica fundamental de la virtualización de servicios. Los servicios simulados están disponibles continuamente bajo el control del equipo de desarrollo, eliminando las dependencias externas que podrían bloquear el progreso del proyecto. Esta disponibilidad permite que los desarrolladores trabajen de manera ininterrumpida independientemente del estado de los servicios externos reales.

La configuración flexible de escenarios habilita la simulación de diferentes condiciones operacionales que serían difíciles o imposibles de reproducir con servicios reales. Los servicios virtualizados pueden configurarse para simular alta latencia, respuestas de error específicas, o comportamientos dinámicos basados en el contenido de las *requests*. Esta capacidad resulta fundamental para hacer testing comprehensivo que cubra no solamente el camino feliz sino también condiciones de error y casos borde.

La reproducibilidad que proporcionan los servicios simulados es esencial para la ejecución de pruebas automatizadas y la validación de comportamientos específicos. A diferencia de los servicios reales que pueden comportarse de manera impredecible debido a factores externos, los servicios simulados proporcionan respuestas consistentes que facilitan la identificación de problemas y la validación de correcciones.

2.2.2 Beneficios de la simulación de servicios

La implementación de Service Virtualization aporta múltiples beneficios a los equipos de desarrollo y QA. La aceleración del desarrollo emerge naturalmente de la capacidad de los desarrolladores para trabajar en paralelo sin esperar a que los servicios externos estén disponibles o sean desarrollados por otros equipos [8]. Esta independencia elimina bloqueos y permite que el trabajo progrese de manera continua.

La calidad del testing mejora sustancialmente mediante la capacidad de simular casos borde y condiciones de error que permiten realizar pruebas más exhaustivas e identificar errores que podrían no ser detectados con servicios reales. La posibilidad de reproducir exactamente las mismas condiciones en cada ejecución de pruebas facilita la validación de correcciones y previene regresiones. [13]

Los costos operacionales se reducen al eliminar los gastos asociados con el uso de servicios externos durante el desarrollo y testing, especialmente relevante para APIs que cobran por transacción [10]. Esta reducción de costos puede ser significativa en organizaciones con equipos grandes o ciclos de testing intensivos.

El aislamiento de pruebas permite que cada equipo trabaje con sus propios datos y configuraciones sin afectar a otros equipos, reduciendo los conflictos y dependencias que caracterizan entornos donde múltiples equipos comparten servicios de testing [14]. Esta independencia operacional mejora la productividad al eliminar la necesidad de coordinación explícita entre equipos para el uso de recursos compartidos.

Los pipelines de integración continua pueden ejecutarse de manera más rápida y confiable al no depender de servicios externos que podrían estar no disponibles o tener tiempos de respuesta variables. Esta confiabilidad es fundamental para prácticas de DevOps donde la retroalimentación rápida sobre la calidad del código es esencial [15].

Sin embargo, las soluciones de Service Virtualization existentes en el mercado presentan limitaciones significativas, particularmente en entornos de desarrollo colaborativo donde múltiples

usuarios necesitan trabajar simultáneamente con configuraciones personalizadas. Esta problemática específica será abordada en detalle en la siguiente sección mediante el análisis de herramientas existentes.

2.3 Análisis de herramientas existentes

Para comprender las limitaciones actuales del mercado de simulación de servicios, se realizó un análisis exhaustivo de las principales herramientas disponibles. Este análisis se centró en evaluar las capacidades de manejo de tráfico, configuración por usuario, y adaptabilidad a entornos de desarrollo colaborativo. Las herramientas seleccionadas representan diferentes enfoques: soluciones locales, servicios cloud, y herramientas integradas en ecosistemas de desarrollo de APIs.

2.3.1 WireMock

WireMock [4] es una herramienta de código abierto diseñada para la creación rápida de servidores mock de APIs REST localmente. Su principal fortaleza radica en la simplicidad de configuración y la facilidad de uso para desarrolladores individuales que necesitan simular APIs de manera local durante el desarrollo.

La operación completamente local garantiza control total sobre las configuraciones y elimina dependencias de servicios externos. Esta característica es particularmente valiosa en entornos donde la conectividad a internet es limitada o donde existen restricciones de seguridad que impiden el uso de servicios externos. WireMock permite configurar respuestas en JSON, XML, y otros formatos, además de simular diferentes códigos de estado HTTP y headers personalizados. Adicionalmente, incluye funcionalidades para generar datos aleatorios mediante código y plantillas, lo cual resulta útil para crear datasets de prueba variados y realistas.

La arquitectura de configuración global representa una limitación fundamental para equipos colaborativos. La herramienta no proporciona mecanismos para que múltiples desarrolladores trabajen con configuraciones independientes simultáneamente, lo cual puede generar conflictos en equipos donde diferentes miembros necesitan simular escenarios distintos [16]. La arquitectura local tampoco escala adecuadamente para entornos de desarrollo complejos donde múltiples servicios necesitan ser simulados de forma coordinada y donde se requiere orquestación entre diferentes componentes simulados.

WireMock Cloud, una versión comercial de la herramienta, aborda algunas de estas limitaciones mediante mayor soporte para equipos distribuidos y capacidades de redireccionamiento dinámico de tráfico. Sin embargo, persiste la ausencia de soporte apropiado para configuraciones específicas por usuario, y el modelo de licenciamiento pago introduce consideraciones presupuestarias que no siempre pueden ser contempladas.

2.3.2 Beeceptor

Beeceptor [6] es una solución cloud que proporciona capacidades de interceptación y modificación de solicitudes HTTP, actuando de manera similar a un proxy. Su enfoque está orientado hacia la inspección de tráfico y la simulación básica de APIs con una configuración web accesible. La herramienta funciona como un proxy HTTP que permite interceptar, inspeccionar y modificar *requests* en tiempo real, proporcionando visibilidad completa del tráfico que pasa a través del sistema. Esta capacidad de interceptación resulta especialmente valiosa para entender el comportamiento de aplicaciones existentes y para investigar problemas de integración.

La generación automática de endpoints mock basándose en el tráfico observado acelera significativamente la configuración inicial y reduce el tiempo necesario para poner en marcha simulaciones básicas. Adicionalmente, la plataforma proporciona herramientas de análisis que permiten entender patrones de uso y comportamiento de las APIs, incluyendo métricas de frecuencia de *requests*, distribución de métodos HTTP, y análisis de payload.

Sin embargo, la plataforma presenta limitaciones significativas para entornos colaborativos. La ausencia de configuraciones granulares por dispositivo o usuario individual puede resultar en conflictos cuando múltiples desarrolladores intentan trabajar con diferentes escenarios simultáneamente, especialmente en equipos grandes donde es común que diferentes miembros necesiten simular comportamientos específicos para sus casos de uso. Esta limitación en aislamiento por usuario representa un obstáculo fundamental para adopción en equipos de desarrollo colaborativo.

El modelo cloud introduce dependencias externas y potenciales problemas de latencia o disponibilidad que pueden afectar el flujo de desarrollo, especialmente problemático en entornos donde la conectividad a internet es inestable. La necesidad de conectividad continua para acceder a la plataforma puede ser restrictiva en ciertos contextos operacionales. Adicionalmente, el modelo de licenciamiento pago requiere consideraciones presupuestarias, junto con el riesgo de que los términos y condiciones de las licencias cambien a lo largo del tiempo.

2.3.3 Postman

Postman incluye funcionalidades de Mock Server que permiten simular contratos de API dentro de su ecosistema [5]. Esta funcionalidad está diseñada principalmente para validar contratos de API y facilitar el desarrollo paralelo entre equipos de frontend y backend. Los Mock Servers se integran nativamente con Collections, Environments, y otras herramientas de Postman, proporcionando un flujo de trabajo cohesivo para equipos que ya utilizan la plataforma para documentación y testing de APIs.

La capacidad de crear mocks basándose en especificaciones OpenAPI o definiciones de Collections garantiza consistencia entre la documentación y las simulaciones. Esta aproximación basada en contratos resulta especialmente valiosa en equipos que siguen metodologías de API-first development, donde la especificación de la API se define antes de su implementación.

No obstante, Postman Mock Server presenta limitaciones significativas para casos de uso que requieren mayor flexibilidad. La ausencia de funcionalidades de proxy inteligente para interceptar y redirigir tráfico real hacia servicios simulados requiere cambios explícitos en las configuraciones de las aplicaciones para apuntar a los endpoints simulados en lugar de los servicios reales. Similar a las otras herramientas analizadas, no ofrece configuraciones específicas por dispositivo o sesión individual, limitando la capacidad de múltiples desarrolladores para trabajar con escenarios personalizados simultáneamente.

El enfoque centrado en contratos, aunque útil para validación, puede ser limitante para casos de uso que requieren simulaciones más dinámicas o basadas en comportamiento real observado, ya que estos contratos no son provistos por los servicios o están desactualizados.

2.3.4 Análisis comparativo

El análisis de estas herramientas revela un patrón común de limitaciones que afectan significativamente su aplicabilidad en entornos de desarrollo colaborativo. La Tabla 1 sintetiza las características principales de las herramientas analizadas, enfocándose en los criterios relevantes para contextos donde múltiples desarrolladores requieren configuraciones independientes y simultáneas.

Criterio	WireMock	Beeceptor	Postman Mock Server
Tipo de despliegue	Local (JAR)	Cloud	Cloud / Local
Aislamiento por usuario/sesión	No soportado	No soportado	No soportado
Proxy híbrido integrado	Basico	Básico	No soportado
Captura automática de tráfico	Limitada	Sí	No
Configuración programática	Sí (Java/JSON)	Web GUI	Collections/OpenAPI
Licenciamiento	OSS / Comercial	Freemium	Freemium
Dependencia de conectividad	No	Sí	Parcial

Tabla 1. Análisis comparativo de herramientas de simulación de servicios

Como se observa en la Tabla 1, ninguna de las herramientas analizadas proporciona aislamiento por sesión de usuario, capacidad identificada como crítica para eliminar conflictos en equipos de desarrollo donde múltiples personas trabajan simultáneamente con diferentes escenarios de prueba. Esta limitación común constituye el vacío principal que motiva el desarrollo de una solución alternativa.

Adicionalmente, ninguna de las herramientas proporciona capacidades nativas de proxy híbrido que permitan manejar tráfico mixto, combinando servicios reales y simulados de manera transparente y configurable por usuario. Esto representa una limitación fundamental que requiere modificaciones en el código de las aplicaciones o configuraciones manuales complejas para alternar entre servicios reales y simulados.

Las soluciones existentes operan con configuraciones globales o, en el mejor de los casos, por proyecto, pero carecen de la capacidad de proporcionar configuraciones específicas por dispositivo o sesión de usuario. Esta limitación resulta especialmente problemática en equipos grandes donde múltiples desarrolladores necesitan trabajar simultáneamente con diferentes escenarios, causando bloqueos y conflictos que reducen la productividad del equipo. Aunque algunas herramientas soportan colaboración básica a través de workspaces compartidos o configuraciones sincronizadas, ninguna resuelve efectivamente el problema de permitir que múltiples desarrolladores trabajen con configuraciones completamente independientes y personalizadas al mismo tiempo.

Estas limitaciones fundamentales justifican la necesidad de una solución más comprehensiva que aborde específicamente los desafíos de entornos de desarrollo distribuidos y colaborativos. La propuesta debe incorporar capacidades de proxy inteligente que permitan operación híbrida transparente, configuraciones granulares por sesión de usuario que eliminen conflictos entre desarrolladores trabajando simultáneamente, y soporte robusto para escenarios complejos que incluya captura automática de tráfico y generación de respuestas. Estos componentes diferenciadores, y la arquitectura que los integra, se detallarán en los capítulos posteriores de esta tesis.

3. Motivación y análisis del problema

En el contexto actual del desarrollo de software, los equipos enfrentan desafíos crecientes relacionados con la dependencia de servicios externos, especialmente en sectores críticos como el bancario y financiero. Este capítulo analiza el problema específico que motivó el desarrollo de *Mimick*,

explorando las limitaciones identificadas en el ecosistema de desarrollo de software y su impacto en la productividad de los equipos.

3.1 Identificación del problema

3.1.1 Dependencias de servicios de terceros en el desarrollo de software

La incorporación de dependencias, tales como APIs de terceros o microservicios, se ha consolidado como una práctica estándar que permite evitar la duplicación de esfuerzos, al reducir la necesidad de diseñar, escribir, testear, depurar y mantener componentes ya resueltos en otros entornos. Esta estrategia introduce también riesgos significativos, que van desde cuestiones de seguridad hasta problemas de disponibilidad y compatibilidad.

En el ámbito bancario, estos problemas se manifiestan con especial intensidad. La disponibilidad de los servicios de terceros es un factor crítico: interrupciones frecuentes o fallas en las APIs generan bloqueos en el desarrollo y retrasos en las entregas, afectando tanto a la productividad de los equipos como a la calidad del software final.

Más allá de la disponibilidad, la latencia representa otro de los desafíos centrales. Los tiempos de respuesta inconsistentes y elevados de los servicios externos afectan de manera directa el desarrollo y las pruebas. El impacto no se limita únicamente al aumento de los tiempos de ejecución de los tests, sino también a la dificultad para distinguir entre problemas propios del código y retrasos ocasionados por los servicios externos.

Finalmente, los servicios reales presentan serias limitaciones al momento de configurar escenarios de prueba no convencionales. En el caso de las aplicaciones financieras, no siempre es posible reproducir en un entorno real situaciones como la existencia de un cliente con múltiples productos financieros de distinta naturaleza. Estas restricciones reducen la capacidad de los equipos de testing para cubrir casos borde y generan brechas en la validación de los sistemas.

3.1.2 Impacto en el flujo de desarrollo

El uso de servicios externos en entornos distribuidos no solo condiciona la calidad del software, sino que también transforma la forma en que los equipos pueden trabajar. Cuando los componentes dependen de APIs o sistemas que no siempre están disponibles —ya sea por fallas, restricciones de acceso o simplemente porque aún no han sido desarrollados— el trabajo se ve interrumpido y se pierde la posibilidad de avanzar en paralelo.

3.2 Impacto en equipos de desarrollo y QA

Uno de los efectos más notorios de estas limitaciones es la pérdida de independencia entre los desarrolladores. Cuando varios equipos deben trabajar con el mismo conjunto de servicios externos, los cambios de configuración realizados por un integrante pueden alterar el comportamiento esperado en las pruebas de otros, lo que genera conflictos y ralentiza el progreso colectivo.

Los equipos de *Quality Assurance* también enfrentan desafíos específicos en este escenario. La generación de datos de prueba adecuados suele ser una tarea compleja, ya que en muchos casos es necesario recurrir a la asistencia de desarrolladores backend o realizar configuraciones manuales en sistemas externos, lo que entorpece la labor de QA. Además, la variabilidad inherente a los servicios externos dificulta la reproducibilidad de errores, un aspecto esencial para asegurar la confiabilidad de los

sistemas. Sin acceso a mecanismos de simulación, la cobertura de pruebas se ve limitada por las restricciones impuestas por los entornos reales. La utilización de mocks, en cambio, permite aislar los tests, controlar las respuestas y reproducir fallas de manera controlada, lo que constituye una ventaja sustancial en términos de eficiencia y exhaustividad de las pruebas [1]

Las consecuencias de estas limitaciones se reflejan directamente en métricas de productividad. El tiempo de ciclo de desarrollo se ve incrementado debido a las esperas ocasionadas por la indisponibilidad de servicios externos. La calidad del software disminuye al quedar sin cubrir escenarios críticos que no pueden reproducirse en entornos reales. Finalmente, la satisfacción de los equipos se ve afectada, ya que la imposibilidad de mantener un flujo de trabajo estable y continuo genera frustración y aumenta la carga cognitiva de los desarrolladores. [14]

3.3 Análisis de mercado y justificación

Las soluciones existentes en el mercado abordan sólo parcialmente estos problemas. Herramientas como *WireMock* se han convertido en referentes del sector y han demostrado ser útiles para reducir la dependencia en APIs inestables, pero requieren configuraciones técnicas complejas y no proporcionan mecanismos efectivos de aislamiento entre usuarios [16]. Por otra parte, servicios en la nube como *Wiremock Cloud* ofrecen interfaces más accesibles para la simulación de APIs, aunque dependen de la conectividad permanente y presentan limitaciones en términos de personalización y costos, especialmente para equipos grandes.

En consecuencia, se hace evidente la necesidad de una solución que combine la independencia de desarrollo con la capacidad de soportar casos propios del dominio bancario. El sector BFSI (Banking, Financial Services and Insurance) presenta exigencias particulares debido a la complejidad de sus sistemas, el alto grado de integración requerido y las estrictas demandas de seguridad. En este contexto, *Mimick* surge como una herramienta orientada a proporcionar entornos de simulación que permitan reproducir casos límite, reducir la dependencia de servicios externos y facilitar un trabajo verdaderamente paralelo entre equipos de desarrollo y QA.

3.4 Objetivos del proyecto y criterios de éxito

A partir del análisis exhaustivo de la problemática identificada y su impacto directo en la productividad del equipo de desarrollo, se establecieron objetivos concretos con criterios medibles que guiarían tanto el diseño de Mimick como la evaluación de su éxito. Estos objetivos surgieron directamente de las limitaciones más críticas documentadas en las secciones anteriores, proporcionando un marco claro para validar que la solución efectivamente resolviera las necesidades del equipo.

- Desarrollo paralelo sin conflictos

Eliminar los bloqueos sistemáticos causados cuando múltiples desarrolladores necesitan simular diferentes estados del mismo servicio externo. La arquitectura de aislamiento por sesión debe permitir que cada desarrollador mantenga configuraciones completamente independientes sin requerir coordinación explícita con otros miembros del equipo.

Los criterios de éxito establecidos para este objetivo incluían la capacidad del sistema para soportar al menos 15 desarrolladores trabajando simultáneamente sin degradación de rendimiento o interferencias entre configuraciones. El rendimiento debía mantenerse consistente con respuestas en menos de 250 milisegundos por request, una mejora sustancial sobre las latencias de 1.5 a 2 segundos observadas en los peores casos con servicios externos reales. El sistema también debía demostrar capacidad para manejar al

menos 100 *requests* por segundo, suficiente para soportar múltiples sesiones de desarrollo y pruebas ejecutándose en paralelo.

- Reproducción ágil de incidentes de producción

Transformar el proceso de diagnóstico de incidentes que frecuentemente requiere días o semanas para reproducir condiciones específicas de producción. Proporcionar mecanismos para capturar, almacenar y recrear comportamientos exactos de servicios externos observados durante incidentes críticos, acelerando significativamente el ciclo de resolución de problemas.

El criterio principal de éxito establecía que cualquier incidente de producción debía poder reproducirse en un entorno controlado dentro del mismo día laboral en que se reportara. Esto representaba una reducción dramática desde los 2 a 10 días que típicamente requería la configuración manual de escenarios complejos en ambientes de desarrollo. La funcionalidad de captura y reproducción de tráfico debía ser lo suficientemente intuitiva para que cualquier miembro del equipo pudiera utilizarla sin requerir conocimiento especializado de la herramienta.

- Viabilización de testing automatizado integral:

Habilitar la implementación de suites de pruebas automatizadas end-to-end que permanecen fuera de alcance debido a la inestabilidad e impredecibilidad de los servicios externos. Proporcionar la estabilidad y determinismo necesarios para ejecutar pruebas integrales de manera confiable, eliminando los falsos negativos y timeouts que comprometen la confianza en los resultados de las pruebas.

Los criterios de éxito para este objetivo establecen alcanzar al menos 50% de cobertura en pruebas end-to-end para casos críticos del negocio. Si bien este porcentaje podría parecer conservador, representaba un avance fundamental desde la situación inicial donde la ejecución confiable de cualquier prueba integral resultaba prácticamente imposible. Las pruebas debían ejecutarse de manera determinística, produciendo resultados idénticos en cada ejecución, eliminando los falsos negativos causados por timeouts o indisponibilidad de servicios externos. Para operaciones administrativas como el encendido y apagado del proxy que no formarían parte del flujo crítico de pruebas, se consideró aceptable una latencia de hasta un segundo.

Estos objetivos y sus criterios asociados no solo proporcionaron dirección clara durante el desarrollo, sino que también establecieron métricas objetivas para evaluar el éxito del proyecto. La definición explícita de estos parámetros desde el inicio ayudó a evitar que se agreguen funcionalidades que, aunque técnicamente interesantes, no contribuían directamente a resolver los problemas fundamentales del equipo.

4. Propuesta de solución

Habiendo identificado las limitaciones de las herramientas existentes y el impacto concreto que estas generan en los equipos de desarrollo y QA, este capítulo presenta a Mimick como una solución integral diseñada específicamente para abordar los problemas descritos en el capítulo anterior. La propuesta se fundamenta en tres pilares diferenciadores que, trabajando de manera conjunta, permiten superar las restricciones identificadas en las soluciones actuales del mercado: un componente de Proxy transparente con capacidades avanzadas de redirección, un sistema de gestión de sesiones por usuario que permite

aislamiento completo entre desarrolladores, y una funcionalidad de grabado automático de tráfico que elimina la necesidad de configuración manual extensiva.

La concepción de Mimick respondió inicialmente a la intención de incorporar técnicas de inteligencia artificial para la generación y optimización automática de simulaciones de servicios. Sin embargo, el proceso de validación empírica reveló que la complejidad introducida por estas técnicas no justificaba los beneficios obtenidos, generando resultados impredecibles y aumentando significativamente los tiempos de procesamiento. Esta experiencia condujo al equipo hacia un enfoque más pragmático, centrado en la captura directa y reproducción fiel de tráfico real, priorizando la confiabilidad y simplicidad por sobre la generación algorítmica. Este cambio tecnológico, que será analizado en profundidad en el capítulo de lecciones aprendidas, resultó en una solución más robusta y mantenible que cumple efectivamente con los objetivos establecidos.

4.1 Visión general de Mimick

Mimick se concibe como una herramienta de uso interno orientada a agilizar los tiempos de desarrollo y minimizar la cantidad de errores en producción. El objetivo principal trasciende la mera simulación de servicios: busca proporcionar a los equipos la capacidad de reproducir rápidamente incidentes o errores encontrados en distintos estadios del desarrollo, permitiendo un diagnóstico más ágil y preciso de problemas que de otra manera requerirían acceso a servicios externos con configuraciones específicas y potencialmente inestables.

La filosofía de diseño de Mimick se fundamenta en el aprovechamiento de una ventaja operacional crítica: al operar dentro de una infraestructura unificada donde todas las aplicaciones respetan una arquitectura común, resulta posible definir una forma consistente de identificar dispositivos y sesiones, garantizando así el comportamiento aislado para cada uno de ellos. Esta característica, que representa el núcleo diferenciador de la herramienta, contrasta marcadamente con las soluciones analizadas en el capítulo dos, donde la ausencia de gestión de sesiones granular constituye una limitación fundamental que genera conflictos entre equipos cuando múltiples desarrolladores necesitan simular diferentes estados del mismo servicio.

Adicionalmente, la incorporación de un componente de Proxy posicionado entre los solicitantes y las dependencias externas habilita la implementación de funcionalidades avanzadas de captura de tráfico. Esta arquitectura permite generar definiciones de servicios de manera automática a partir del análisis del tráfico capturado, eliminando la necesidad de que los equipos de desarrollo proporcionen especificaciones por adelantado.

Mimick fue diseñado con dos usuarios en mente: los equipos de desarrollo y los de QA. Para el equipo de desarrollo, la herramienta ofrece la posibilidad de probar servicios que todavía no están implementados, configurar casos borde para validar escenarios específicos, y disponer de latencia constante para realizar pruebas de performance confiables.

Por su parte, el equipo de QA puede utilizar Mimick para construir pruebas de regresión que validen que el comportamiento preexistente no sea alterado por nuevas modificaciones. Estas pruebas pueden ser automatizadas y ejecutadas en paralelo gracias a la naturaleza del aislamiento por sesión que implementa Mimick, característica que representa una ventaja significativa sobre las alternativas evaluadas.

4.2 Componentes diferenciadores

La arquitectura de Mimick se sustenta en tres componentes principales que, operando de manera integrada, conforman una solución comprehensiva para la simulación de servicios con gestión de sesiones. Cada uno de estos componentes aborda limitaciones específicas identificadas en las herramientas existentes y, en conjunto, proporcionan capacidades que no se encuentran disponibles en las alternativas evaluadas en el capítulo dos.

4.2.1 Proxy

El componente de Proxy constituye el punto de entrada para todo el tráfico que sale del servidor hacia los servicios que se desea emular. Esta posición le da al Proxy control total sobre cada request que abandona el sistema, habilitando funcionalidades tanto de redirección como de observabilidad del tráfico.

El Proxy está compuesto por dos elementos fundamentales que trabajan en conjunto. Por un lado, una API de configuración permite definir las reglas de redireccionamiento de tráfico de manera programática, proporcionando flexibilidad para establecer políticas granulares según las necesidades de cada equipo. Por otro lado, un componente basado en NGINX actúa como proxy transparente, interceptando el tráfico, grabándolo cuando corresponde, y redigiéndolo a Mimick según las reglas configuradas. La elección de NGINX como motor subyacente se fundamenta en su probada capacidad para manejar altos volúmenes de tráfico con baja latencia [17], características críticas para minimizar el impacto en el rendimiento percibido durante el desarrollo y testing.

Las capacidades de configuración del Proxy operan en múltiples niveles de granularidad. Es posible especificar reglas de redireccionamiento por identificador de sesión, lo cual permite que diferentes desarrolladores trabajen simultáneamente con configuraciones completamente distintas del mismo servicio. Esta granularidad se extiende hasta el nivel de URL individual, permitiendo que todos los *requests* dirigidos a una URL específica desde un dispositivo determinado sean redirigidos a Mimick, mientras que el tráfico hacia otras URLs del mismo servicio continúe alcanzando el endpoint real. Esta flexibilidad en la configuración aborda directamente el problema de configuración global compartida identificado en herramientas como WireMock o Postman, donde modificaciones en la configuración de un desarrollador impactan inevitablemente en el trabajo de otros miembros del equipo.

Respecto a la transparencia operacional, el Proxy requiere una modificación mínima en la configuración de los clientes: los servicios deben apuntar sus *requests* a las URLs expuestas por el proxy en lugar de directamente a los servicios reales. Por ejemplo, si un servicio autentica contra auth.com/api/v1/users, el cliente debe configurarse para apuntar a proxy.auth.com/api/v1/users. La forma en la que los equipos trabajan en la organización ya contempla que todos los hostnames estén definidos como variables de configuración, por lo cual esta modificación resulta trivial de implementar y no requiere cambios en el código de la aplicación. Esta característica minimiza la barrera de entrada para la adopción de Mimick, un factor crítico para el éxito de cualquier herramienta que busque integrarse en flujos de trabajo establecidos.

4.2.2 Manejo de sesión de usuarios

El manejo de sesiones por usuario representa el elemento diferenciador más significativo de Mimick respecto a las herramientas existentes. En el contexto de esta solución, manejo de sesión se refiere específicamente a la capacidad de identificar de manera única cada interacción usuario-sistema y asociar a cada una de estas interacciones un estado completamente independiente. Esta arquitectura permite que múltiples usuarios configuren respuestas distintas para el mismo endpoint simultáneamente, y que cada

uno opere en un entorno completamente aislado de los demás, eliminando así los conflictos que caracterizan a las soluciones con configuración global compartida.

La identificación de cada sesión se implementa mediante un identificador único que se asigna en el momento de creación de la sesión y se transmite en un header específico con cada request. Este mecanismo de identificación, si bien simple en su concepción, resulta suficientemente robusto para los casos de uso objetivo y tiene la ventaja de no requerir infraestructura compleja de autenticación o gestión de tokens.

El aislamiento por sesión habilita escenarios de uso que resultan imposibles o extremadamente complejos con las herramientas evaluadas en el capítulo dos. Por ejemplo, un desarrollador puede configurar y ejecutar pruebas de casos borde sobre un servicio específico, mientras simultáneamente otro desarrollador del mismo equipo realiza pruebas de performance sobre ese mismo servicio con una configuración completamente diferente, y el resto de los usuarios continúan utilizando el servicio real sin ninguna interferencia. Este nivel de independencia elimina uno de los puntos de fricción más significativos para este proyecto: la necesidad de coordinación explícita para el uso de recursos compartidos de testing.

El estado asociado a cada sesión se persiste en una base de datos junto con el identificador de sesión correspondiente. Esta decisión de diseño garantiza que las configuraciones sobrevivan a reinicios del sistema y permite que los usuarios retomen su trabajo exactamente donde lo dejaron. Adicionalmente, se almacena información sobre el escenario configurado, lo cual facilita la capacidad de restaurar estados de sesión específicos. Esta funcionalidad resulta particularmente valiosa cuando se trabaja en la reproducción de errores complejos que requieren configuraciones específicas de múltiples servicios: el desarrollador puede guardar el estado completo de su entorno de simulación y compartirlo con otros miembros del equipo o con el personal de soporte, eliminando así el proceso tedioso y propenso a errores de reconfiguración manual.

La gestión persistente del estado de sesión también habilita patrones avanzados de testing como la creación de "ambientes congelados" que replican exactamente las condiciones bajo las cuales se manifestó un problema en un entorno superior. La capacidad de capturar, almacenar y restaurar estados completos de interacción con servicios externos constituye una herramienta poderosa para el análisis post-mortem de incidentes y para la construcción de casos de prueba que validen el comportamiento del sistema ante condiciones específicas difíciles de reproducir con servicios reales.

4.2.3 Grabado de Tráfico

La funcionalidad de grabado de tráfico completa la propuesta de valor de Mimick al atacar uno de los problemas más importantes que tenemos para garantizar la adopción de herramientas de simulación de servicios: la necesidad de configuración manual extensiva. El grabado de tráfico captura de manera automática y transparente todo el contenido de los *requests* y respuestas que transitan por el Proxy, y cuando Mimick está configurado para grabar tráfico para un dispositivo específico, este tráfico capturado es procesado automáticamente para generar las definiciones de mock correspondientes.

El mecanismo de grabado opera de manera integrada con el componente de Proxy. Cuando un request atraviesa el Proxy, esté captura tanto la solicitud como la respuesta completa, incluyendo headers, payload y metadatos relevantes. Si la sesión asociada al request tiene habilitado el modo de grabado, esta información es enviada a Mimick para su procesamiento y almacenamiento. Este procesamiento genera automáticamente las especificaciones necesarias para reproducir el comportamiento observado, creando efectivamente un mock que refleja fielmente las características del servicio real.

La ventaja principal del grabado automático sobre la configuración manual radica en la reducción dramática del tiempo requerido para poner en funcionamiento un mock operativo. En muchos casos, particularmente cuando se trabaja con servicios legacy o con APIs poco documentadas, no existe una especificación formal del comportamiento del servicio, lo cual hace extremadamente difícil y propenso a errores el proceso de configurar un mock manualmente.

El tráfico grabado no constituye un artefacto estático inmutable. A través de la interfaz de Mimick, los usuarios pueden editar y modificar las definiciones generadas automáticamente para ajustarlas a sus necesidades específicas. Esta capacidad híbrida, que combina generación automática con refinamiento manual, proporciona un balance óptimo entre velocidad de configuración y control granular sobre el comportamiento simulado. Un desarrollador puede iniciar rápidamente con una simulación generada automáticamente que captura el comportamiento general del servicio, y posteriormente refinar aspectos específicos para cubrir casos borde o condiciones de error que no fueron observadas en el tráfico capturado.

La relación entre el grabado de tráfico, el Proxy y las sesiones de usuario establece un flujo de trabajo cohesivo y eficiente. Cuando un usuario desea configurar un nuevo flujo en Mimick, el proceso consiste simplemente en habilitar el grabado de tráfico para su sesión específica y utilizar la aplicación normalmente, interactuando con los servicios reales a través del Proxy. El sistema captura automáticamente todo el tráfico relevante, lo procesa, y genera los mocks necesarios, los cuales quedan asociados a la sesión del usuario. Este flujo minimiza la fricción cognitiva y el tiempo requerido para comenzar a trabajar con simulaciones, acelerando el acceso a capacidades avanzadas de testing que anteriormente requerían conocimiento técnico especializado sobre el funcionamiento interno de herramientas de mocking.

La configuración del grabado se realiza a nivel de sesión, lo cual significa que diferentes usuarios pueden tener distintas políticas de grabado activas simultáneamente sin interferencia entre ellos. Esta característica se alinea con el principio arquitectónico fundamental de aislamiento por sesión que permea toda la solución, y habilita escenarios donde un usuario puede estar capturando tráfico para generar nuevos mocks mientras otros usuarios están consumiendo mocks ya existentes para sus pruebas, todo ello sin requerir coordinación explícita o generar conflictos operacionales.

En su conjunto, estos tres componentes conforman una solución integral que aborda las limitaciones identificadas en las herramientas existentes mediante una arquitectura que privilegia el aislamiento, la flexibilidad y la facilidad de uso. El Proxy proporciona control y visibilidad sobre el tráfico, el manejo de sesiones garantiza independencia entre usuarios, y el grabado de tráfico elimina barreras de entrada para la configuración de simulaciones complejas. El capítulo siguiente detallará la arquitectura técnica que sustenta estos componentes y las decisiones de diseño que permitieron materializarlos en un sistema funcional y escalable.

5. Arquitectura del sistema

Habiendo establecido la propuesta conceptual de Mimick y sus componentes diferenciadores en el capítulo anterior, este capítulo describe la arquitectura técnica que materializa dicha propuesta. El diseño arquitectónico responde a requisitos específicos de flexibilidad, escalabilidad y facilidad de mantenimiento. La arquitectura resultante refleja además el proceso iterativo de desarrollo descrito en el capítulo seis, incorporando aprendizajes de las fases tempranas del proyecto y ajustes motivados por validación empírica en entornos reales.

5.1 Diseño arquitectónico general

La arquitectura de Mimick se estructura en torno a dos componentes principales que operan de manera coordinada pero mantienen responsabilidades claramente diferenciadas: el Proxy, encargado de la interceptación y redirección de tráfico, y el Mocking Engine, responsable de la gestión y servicio de las simulaciones. Esta separación arquitectónica responde a principios fundamentales de diseño de software que promueven la modularidad y la separación de concerns, permitiendo que cada componente evolucione de manera independiente y facilitando tanto el mantenimiento como la extensibilidad del sistema.

La comunicación entre componentes se establece mediante APIs REST, adoptando un enfoque arquitectónico que privilegia la interoperabilidad y el bajo acoplamiento. Esta decisión de diseño se alinea con prácticas contemporáneas en el desarrollo de sistemas distribuidos, donde la utilización de interfaces HTTP estándar facilita la integración entre componentes heterogéneos y simplifica el testing y debugging de las interacciones. La adopción de REST como protocolo de comunicación también habilita potenciales extensiones futuras del sistema, como la incorporación de nuevos componentes o la integración con herramientas externas, sin requerir modificaciones sustanciales en la arquitectura existente.

Complementando estos dos componentes centrales, Mimick incorpora una interfaz de usuario web que actúa como punto de entrada para todas las operaciones de gestión del sistema. Esta interfaz centraliza las interacciones del usuario con tanto el Proxy como el Mocking Engine, proporcionando una experiencia cohesiva que abstrae la complejidad de la arquitectura distribuida subyacente.

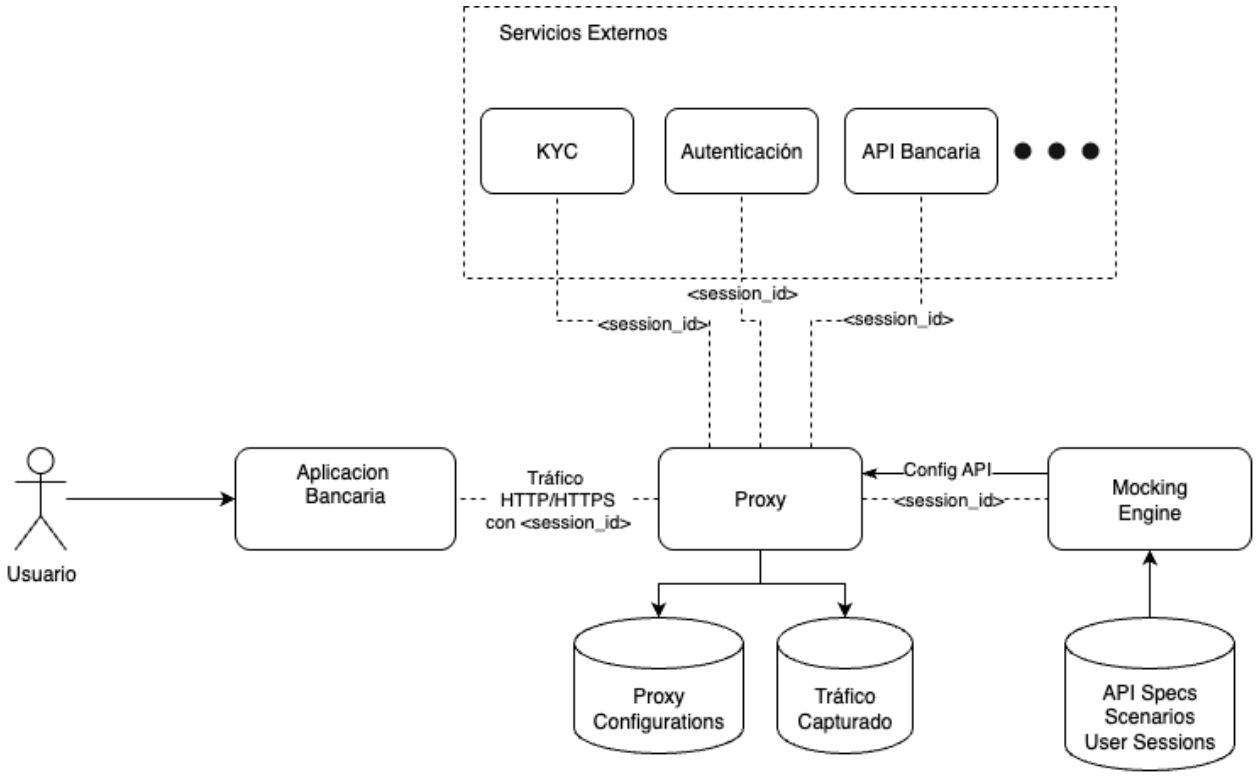

Figura 1. Arquitectura general de Mimick mostrando la relación entre el Proxy, el Mocking Engine, y las distintas bases de datos. Los flujos de tráfico proveniente de interacciones de usuarios con la aplicación bancaria se marcan con líneas punteadas.

El flujo general de una request que atraviesa el sistema ilustra la orquestación entre estos componentes. Cuando una aplicación cliente realiza una solicitud a un servicio externo, el tráfico es interceptado por el Proxy de acuerdo a las reglas de redirección configuradas. El Proxy evalúa la request contra sus criterios de matching, considerando el hostname destino, el path solicitado, y el header de identificación de sesión. Si la evaluación determina que la request debe ser simulada, el Proxy la redirige al Mocking Engine, el cual consulta sus especificaciones almacenadas para determinar la respuesta apropiada según el contexto del usuario y el escenario configurado. La response generada retorna al Proxy, quien la devuelve a la aplicación cliente de manera transparente, manteniendo la ilusión de que la comunicación se estableció con el servicio real.

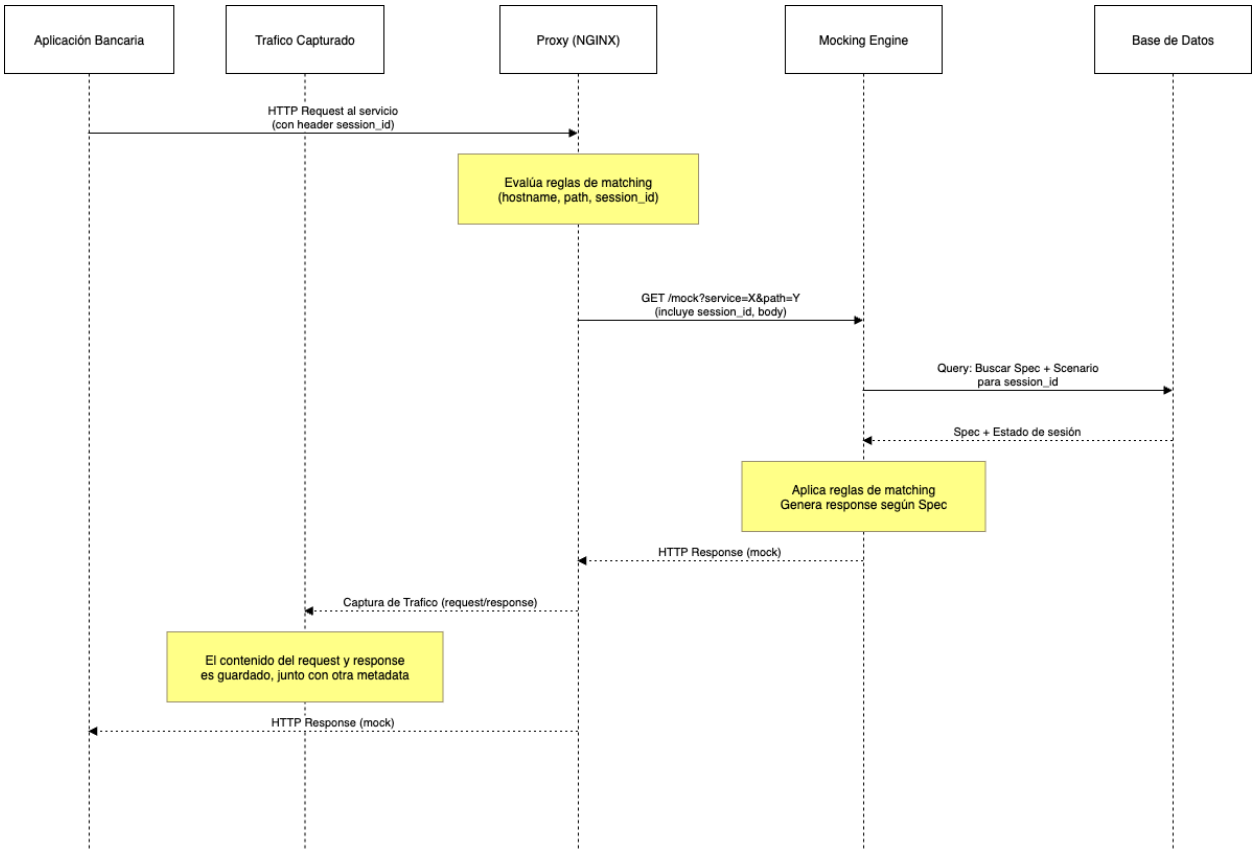

Figura 2. Diagrama de secuencia mostrando el flujo completo de una request desde la aplicación cliente hasta la obtención de una respuesta simulada. Se detallan las interacciones entre Cliente, Proxy, Mocking Engine y Base de Datos.

Este diseño arquitectónico permite que el sistema opere con mínima intrusión en las aplicaciones cliente, requiriendo únicamente que éstas apunten sus *requests* a las URLs expuestas por el Proxy en lugar de directamente a los servicios reales. Esta característica de transparencia operacional constituye uno de los aspectos más valiosos de la arquitectura, minimizando la fricción en la adopción de la herramienta y

permitiendo que los equipos de desarrollo integren Mimick en sus flujos de trabajo existentes sin necesidad de modificaciones sustanciales en su código base.

La decisión de mantener el Proxy y el Mocking Engine como componentes separados pero coordinados se fundamenta en consideraciones tanto técnicas como operacionales. Desde una perspectiva técnica, esta separación permite optimizar cada componente para su responsabilidad específica: el Proxy se enfoca en rendimiento y capacidad de procesamiento de tráfico, mientras que el Mocking Engine puede priorizar flexibilidad y riqueza funcional en la gestión de simulaciones. Desde una perspectiva operacional, la arquitectura facilita estrategias de deployment diferenciadas, permitiendo por ejemplo escalar horizontalmente el Proxy para manejar mayores volúmenes de tráfico sin necesidad de replicar el Mocking Engine, o viceversa.

5.2 Componentes principales: Proxy, APIs y interfaz web

La materialización de la arquitectura descrita se realiza a través de cinco componentes técnicos que implementan las responsabilidades distribuidas del sistema. Cada uno de estos componentes fue seleccionado y diseñado considerando criterios de madurez tecnológica, facilidad de mantenimiento, y alineación con las capacidades del equipo de desarrollo.

5.2.1 Proxy basado en NGINX y OpenResty

El componente de Proxy se implementa utilizando OpenResty [18], una distribución extendida de NGINX que incorpora el motor de scripting LuaJIT. La elección de esta tecnología responde a múltiples consideraciones que la posicionan como la opción más adecuada para los requisitos del sistema. NGINX ha demostrado ser una solución madura y ampliamente probada en entornos de producción de alta demanda, ofreciendo capacidades sobresalientes de procesamiento de tráfico HTTP/HTTPS con consumo eficiente de recursos.

La extensión OpenResty agrega capacidades programáticas mediante Lua, habilitando la implementación de lógica de matching y redirección compleja directamente en el contexto del procesamiento de *requests*. Esta capacidad resulta fundamental para materializar los requisitos de matching granular descritos en el capítulo cuatro, donde la decisión de redirección debe considerar no solamente el hostname y path, sino también información contenida en headers específicos para implementar el aislamiento por sesión de usuario.

El carácter de software libre y código abierto de NGINX constituye otro factor relevante en la decisión. Esta característica elimina dependencias de licenciamiento comercial que podrían generar restricciones presupuestarias o contractuales, y proporciona transparencia total sobre el funcionamiento interno del componente, facilitando debugging y optimizaciones cuando sea necesario. La existencia de una comunidad activa de usuarios y desarrolladores garantiza además acceso a documentación extensiva, foros de discusión técnica, y actualizaciones de seguridad y funcionalidad.

La configuración del Proxy se gestiona mediante una aproximación híbrida que combina persistencia de configuración en base de datos con generación dinámica de archivos de configuración y scripts. Una API REST implementada en Python permite a los usuarios especificar servicios a gestionar, políticas de redirección, y configuraciones de grabado de tráfico. Esta información se persiste en una base de datos SQLite dedicada al Proxy, separada de la base de datos principal del sistema para mantener independencia operacional del componente. Luego se lee esta base de datos y se generan los archivos de configuración de NGINX y scripts Lua correspondientes, los cuales son aplicados mediante recarga del servidor.

Esta arquitectura de configuración proporciona balance entre flexibilidad de gestión mediante API y rendimiento de ejecución. La generación de archivos de configuración estáticos permite que NGINX opere con su máximo rendimiento durante el procesamiento de tráfico, sin necesidad de consultas a base de datos en el path crítico de cada request. Simultáneamente, la exposición de una API de configuración habilita la automatización y gestión programática del Proxy, facilitando casos de uso donde las políticas de redirección deben ser ajustadas dinámicamente en respuesta a cambios en el entorno de testing.

El matching de tráfico opera evaluando tres dimensiones de información: el hostname destino, el path de la request, y un header específico que identifica la sesión del usuario. Esta evaluación multidimensional se implementa mediante scripts Lua que son ejecutados por OpenResty en el contexto del procesamiento de cada request. Los scripts consultan estructuras de datos en memoria que representan las reglas de matching configuradas, determinando si una request particular debe ser redirigida al Mocking Engine o permitida continuar hacia el servicio real. La utilización de estructuras de datos en memoria garantiza que el overhead de evaluación de matching sea mínimo, manteniendo latencias bajas incluso bajo cargas significativas de tráfico.

El Proxy maneja completamente el cifrado TLS/SSL, actuando como punto de terminación para conexiones seguras tanto desde los clientes como hacia los servicios reales o el Mocking Engine. Los certificados para cada servicio gestionado se configuran en los archivos de configuración de NGINX, permitiendo que el Proxy presente certificados válidos a los clientes y valide los certificados de los servicios backend. Esta arquitectura de terminación TLS centralizada simplifica la gestión de certificados y habilita capacidades de inspección de tráfico cifrado, necesarias para implementar las funcionalidades de grabado descritas en el capítulo cuatro.

5.2.2 Proxy API

La Proxy API constituye la interfaz programática mediante la cual otros componentes del sistema y potencialmente herramientas externas pueden configurar y gestionar el comportamiento del Proxy. Implementada en Python utilizando el framework Flask, esta API proporciona endpoints para registrar nuevos servicios bajo gestión del Proxy, definir políticas de redirección de tráfico, y controlar el grabado de interacciones. La elección de Flask se fundamenta en su simplicidad y peso ligero, características que lo posicionan como framework ideal para APIs que no requieren la complejidad de frameworks full-stack más pesados.

La funcionalidad de la Proxy API se organiza en tres áreas principales de responsabilidad. La primera comprende la gestión de servicios, permitiendo registrar nuevos servicios externos que serán intermediados por el Proxy. Esta funcionalidad habilita operaciones de alta de servicios especificando hostnames, puertos, certificados TLS cuando corresponda, y metadatos descriptivos. La segunda área abarca la configuración de políticas de redirección, donde se especifican las reglas que determinan qué tráfico debe ser redirigido al Mocking Engine y bajo qué condiciones. Estas políticas operan en el nivel de granularidad requerido para implementar el aislamiento por sesión, permitiendo especificar que *requests* provenientes de ciertos identificadores de sesión sean redirigidos mientras que otros hacia el mismo servicio continúen al endpoint real. La tercera área comprende el control de grabado de tráfico, habilitando y deshabilitando la captura de interacciones para sesiones específicas.

La comunicación entre la Proxy API y el Proxy NGINX se implementa mediante un patrón de configuración indirecta a través de base de datos. Cuando la API recibe una request para modificar configuración, valida los parámetros recibidos y persiste los cambios en la base de datos SQLite dedicada del Proxy. Luego, el proceso que maneja los Proxy NGINX lee esta base de datos y genera los archivos de configuración de NGINX y scripts Lua correspondientes. Una vez generados estos archivos, el proceso

emite un comando de recarga a NGINX, el cual aplica la nueva configuración sin interrumpir el procesamiento de tráfico en curso. Esta arquitectura de configuración asíncrona desacopla la API de la operación del Proxy, permitiendo que ambos componentes operen y escalen independientemente.

5.2.3 API Mimick

La API Mimick constituye el núcleo del Mocking Engine, implementando toda la lógica de negocio relacionada con la gestión de simulaciones y el servicio de respuestas mockeadas. Al igual que la Proxy API, se implementa en Python utilizando Flask, decisión que mantiene consistencia tecnológica en el stack backend del sistema y facilita compartición de código y patrones entre ambas APIs.

Las responsabilidades de la API Mimick abarcan múltiples dominios funcionales que en conjunto materializan las capacidades de simulación del sistema. La gestión de Specifications constituye la primera área de responsabilidad. Una Specification (Spec a partir de ahora), representa la definición completa de cómo debe responderse a una request particular. Cada Spec incluye reglas de matching que determinan cuándo aplica, y la definición de la response a retornar cuando se produce un match. Las reglas de matching operan considerando diversos aspectos de la request: path, método HTTP, headers específicos, y en casos más complejos, análisis del body de la request.

La definición de la *response* en una Spec incluye no solamente el body a retornar, sino también el código de estado HTTP, headers de respuesta, y potencialmente delays artificiales para simular latencias específicas. Esta riqueza en la especificación de respuestas habilita la simulación de comportamientos complejos y el testing de casos borde que serían difíciles o imposibles de reproducir con servicios reales.

La segunda área de responsabilidad comprende la gestión de Scenarios. Un Scenario representa un conjunto cohesivo de configuraciones de Specs que en conjunto definen un comportamiento específico del sistema a simular. Por ejemplo, un Scenario denominado "Usuario Bloqueado" podría incluir modificaciones a múltiples Specs que en conjunto simulan el comportamiento de diversos servicios cuando operan sobre una cuenta de usuario que ha sido bloqueada por razones de seguridad. Los Scenarios son entidades compartidas entre usuarios, habilitando colaboración y reutilización de configuraciones. Un desarrollador puede crear un Scenario para un caso de uso específico y otros miembros del equipo pueden activar ese mismo Scenario en sus propias sesiones, evitando duplicación de esfuerzo en la configuración.

La tercera área fundamental es la gestión del Estado de cada sesión de usuario. El Estado representa la configuración activa para un identificador de sesión específico, incluyendo qué Scenario está asignado, qué configuraciones específicas del Proxy aplican para esa sesión, y potencialmente overrides personalizados de Specs que son exclusivos de esa sesión. El Estado se persiste en la base de datos como una estructura JSON que captura toda la información necesaria para reproducir exactamente el comportamiento configurado. Esta representación flexible permite que cada sesión mantenga configuraciones arbitrariamente complejas sin requerir modificaciones al esquema de la base de datos.

La cuarta responsabilidad clave de la API Mimick es actuar como proxy de las operaciones del Proxy que requieren exposición a usuarios finales. Dado que el Proxy no posee interfaz de usuario propia, todas sus operaciones se realizan a través de la UI de Mimick. La API Mimick expone endpoints que internamente invocan la Proxy API, proporcionando una fachada unificada que simplifica la interacción del usuario con el sistema distribuido.

La quinta funcionalidad que cubre la API Mimicking es la de implementar además un sistema comprehensivo de auditoría mediante la entidad RequestLog, que captura información detallada de cada request procesada. Cada vez que la API evalúa una request contra las Specs configuradas para una sesión, genera un registro que incluye el *session_id* solicitante, la Spec que produjo match (si existe), el path y

método de la request, headers relevantes, el body de la request, el código de estado retornado, y la latencia de procesamiento.

Esta funcionalidad cumple dos propósitos fundamentales. En primer lugar, proporciona trazabilidad completa de qué respuestas exactas sirvió Mimick para cada sesión, facilitando debugging cuando el comportamiento observado difiere del esperado. Los desarrolladores pueden consultar los RequestLogs de su sesión para verificar exactamente qué Spec matcheó y qué respuesta se retornó, eliminando ambigüedad sobre el comportamiento del sistema.

En segundo lugar, y quizás más crítico, los RequestLogs permiten detectar *requests* que no encontraron match con ninguna Spec configurada. Cuando una request llega a Mimick pero ninguna regla de matching aplica, el sistema igualmente genera un RequestLog con *spec_id* nulo, marcando explícitamente que esa request no pudo ser servida. Esta capacidad resulta invaluable para identificar gaps en la configuración de mocks: si un desarrollador observa que su aplicación no funciona correctamente y consulta los RequestLogs, puede descubrir rápidamente que ciertos endpoints no tienen Specs configuradas, guiando la creación de las simulaciones faltantes.

La persistencia de RequestLogs en la base de datos permite además análisis histórico de patrones de uso, identificación de Specs más frecuentemente utilizadas, y potencialmente optimización de configuraciones basándose en tráfico real observado.

Finalmente, la API Mimick implementa la funcionalidad de generación automática de mocks a partir de tráfico grabado. Cuando el Proxy captura interacciones en modo de grabado, envía la información de *requests* y respuestas a la API Mimick. Esta procesa el tráfico capturado, identificando patrones y generando Specs correspondientes que reproducen el comportamiento observado. El procesamiento incluye normalización de datos, identificación de valores parametrizables, y agrupación de interacciones relacionadas. Esta capacidad de autogeneración constituye uno de los diferenciadores más significativos de Mimick, reduciendo dramáticamente el tiempo necesario para configurar simulaciones de servicios complejos.

5.2.4 Base de Datos PostgreSQL

La persistencia de datos en Mimick se gestiona mediante PostgreSQL, sistema de gestión de base de datos relacional que ha demostrado alta confiabilidad y rendimiento en innumerables despliegues de producción. PostgreSQL ofrece conformidad completa con estándares SQL, capacidades avanzadas de integridad referencial, soporte para tipos de datos complejos incluyendo JSON nativo, y un modelo de extensibilidad que permite incorporar funcionalidades adicionales cuando sea necesario.

La facilidad de uso de PostgreSQL, respaldada por documentación extensa y herramientas de administración maduras, constituyó factor importante en la decisión. La existencia de una comunidad activa y el carácter de código abierto del proyecto garantizan acceso a soporte comunitario, actualizaciones regulares, y ausencia de costos de licenciamiento que podrían representar restricciones presupuestarias. Estas características alinean PostgreSQL con los principios de adopción de tecnologías abiertas y probadas que guían las decisiones técnicas del proyecto.

El modelo de datos almacena todas las entidades del dominio de Mimick: Specs con sus reglas de matching y definiciones de response, Scenarios que agrupan configuraciones relacionadas, información de sesiones de usuario incluyendo sus Estados activos, configuraciones del Proxy, y metadatos de auditoría que permiten trazabilidad. El diseño del schema privilegia normalización para garantizar consistencia e integridad referencial, mientras que utiliza selectivamente desnormalización mediante columnas JSON para almacenar estructuras de configuración complejas que exhiben alta variabilidad en su formato.

La gestión del Estado de cada sesión ilustra esta aproximación híbrida. Si bien información estructurada sobre la sesión como identificador, usuario propietario, timestamps de creación y modificación, y referencias a Scenarios asignados se almacena en columnas tipadas apropiadamente, la configuración específica que define el comportamiento de la sesión se persiste como documento JSON. Esta decisión reconoce que el Estado puede incluir configuraciones arbitrariamente complejas y variables que serían costosas de modelar mediante relaciones tradicionales, mientras que el formato JSON proporciona la flexibilidad necesaria sin sacrificar capacidades de consulta dado el soporte nativo de PostgreSQL para operaciones sobre datos JSON.

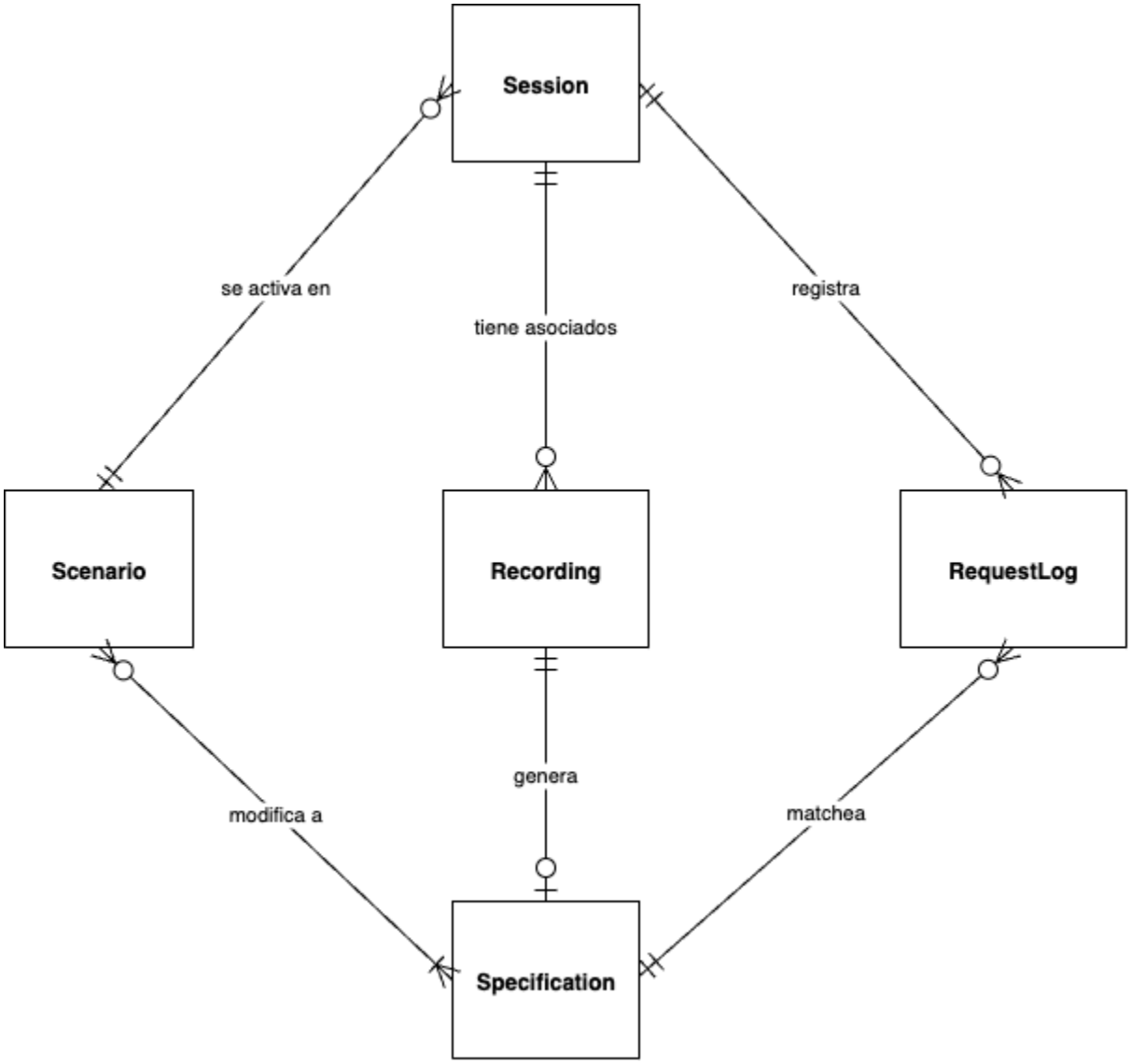

Figura 3. Diagrama del modelo de datos de Mimick mostrando las principales entidades (Specification, Scenarios, Recording, Sessions, RequestLog) y sus relaciones

La base de datos opera como fuente única de verdad para el estado del sistema, garantizando consistencia incluso en escenarios donde múltiples instancias de componentes o múltiples usuarios realizan operaciones concurrentes.

5.2.5 Interfaz Web

La interfaz de usuario de Mimick se implementa como aplicación web que combina un backend basado en Flask con un frontend implementado mediante TypeScript y React. Esta arquitectura híbrida aprovecha las fortalezas de cada tecnología: Flask proporciona una capa de enrutamiento simple y eficiente para servir la aplicación y actuar como backend, mientras que React habilita construcción de interfaces interactivas ricas mediante renderizado del lado del cliente.

La decisión de utilizar una implementación propietaria de client-side rendering en lugar de frameworks establecidos responde a requisitos específicos del entorno de desarrollo de la organización y permite optimizaciones particulares para el caso de uso de Mimick. El uso de TypeScript agrega seguridad de tipos en tiempo de compilación, reduciendo categorías enteras de errores potenciales y mejorando la mantenibilidad del código frontend.

La interfaz web proporciona acceso completo a todas las funcionalidades de gestión del sistema. Los usuarios pueden crear, modificar y eliminar Specs, definiendo para cada una las reglas de matching y las respuestas a retornar. La interfaz de edición de Specs proporciona capacidades avanzadas como validación de sintaxis para payloads JSON y XML, preview de respuestas, y testing inline de reglas de matching. Los usuarios pueden igualmente gestionar Scenarios, agrupando Specs relacionadas y asignándoles nombres descriptivos que faciliten su identificación y reutilización. La interfaz de gestión de sesiones permite a cada usuario visualizar y modificar su Estado activo, activando o desactivando Scenarios, configurando políticas de redirección del Proxy específicas para su sesión, y visualizando el tráfico que llega a Mimick a traves de los RequestLogs.

La visualización de tráfico grabado constituye una funcionalidad particularmente valiosa de la interfaz. Cuando un usuario tiene habilitado el modo de grabado para su sesión, puede acceder a una vista que presenta todas las interacciones capturadas, mostrando para cada una el request completo recibido y la response retornada por el servicio real. La interfaz permite filtrar y buscar dentro del tráfico capturado, facilitando identificación de interacciones específicas de interés. Desde esta vista, el usuario puede seleccionar interacciones y solicitar la generación automática de Specs correspondientes, iniciando el proceso de transformación de tráfico real en mocks reutilizables.

La autenticación en la aplicación web se implementa mediante integración con el sistema de autenticación propietario de la organización, el cual opera mediante tokens JWT. Cuando un usuario accede a la aplicación, es redirigido al sistema de autenticación organizacional donde ingresa sus credenciales. Tras autenticación exitosa, el sistema emite un token JWT que es retornado a la aplicación web de Mimick. Este token se incluye en todas las *requests* subsecuentes a las APIs backend, donde es validado para verificar la identidad del usuario y sus permisos. Esta arquitectura de autenticación delegada evita que Mimick deba implementar y mantener su propio sistema de gestión de usuarios y credenciales, reduciendo complejidad y surface area de seguridad del sistema.

5.4 Análisis de riesgos

El diseño de cualquier sistema de software implica navegación de trade-offs y toma de decisiones bajo incertidumbre sobre comportamiento futuro. El análisis de riesgos arquitectónicos constituye práctica fundamental para identificar amenazas potenciales a los objetivos del sistema y desarrollar estrategias de mitigación apropiadas. Esta sección documenta los principales riesgos identificados durante el diseño de Mimick y las decisiones tomadas para gestionarlos.

Una de las decisiones arquitectónicas más significativas concernió la topología de deployment del Proxy. Se evaluaron dos alternativas fundamentales: un Proxy centralizado que sirva a múltiples clientes o

ambientes, versus múltiples instancias descentralizadas del Proxy donde cada cliente o ambiente mantiene su propia instancia. El enfoque centralizado ofrecería ventajas en términos de simplicidad operacional y consolidación de recursos, requiriendo administración de una única instancia y permitiendo compartición de capacidad de procesamiento entre múltiples tenants. Sin embargo, este enfoque introduce riesgos significativos de punto único de falla y acoplamiento entre ambientes que idealmente deberían estar aislados.

La decisión final privilegió el enfoque descentralizado, donde cada cliente o ambiente mantiene su propia instancia del Proxy. Esta arquitectura maximiza el aislamiento entre ambientes, garantizando que problemas en la instancia del Proxy de un cliente no afecten a otros. El deployment descentralizado también simplifica consideraciones de seguridad y control de acceso, dado que cada instancia del Proxy opera dentro del perímetro de seguridad de su cliente o ambiente específico. La relativa simplicidad del Proxy y su bajo overhead de recursos hacen que la replicación de instancias no represente costo prohibitivo, mientras que los beneficios en términos de aislamiento y resiliencia justifican ampliamente esta decisión arquitectónica.

El rendimiento del Proxy constituyó otro riesgo evaluado cuidadosamente. Como componente que intercepta todo el tráfico hacia servicios externos, el Proxy debe operar con latencia mínima para no degradar la experiencia de desarrollo de los usuarios. La evaluación de alternativas consideró diversos productos y tecnologías, culminando en la selección de OpenResty precisamente por su rendimiento comprobado. La experiencia empírica durante el desarrollo y deployment confirma que OpenResty no agrega latencia significativa al procesamiento de *requests*, manteniendo overhead en el orden de milisegundos incluso bajo cargas moderadas. Esta característica resulta crítica para garantizar que la introducción de Mimick en el flujo de desarrollo no genere frustración por degradación de performance.

La concurrencia en el acceso a APIs de configuración del Proxy representó desafío técnico identificado tempranamente en el proyecto. En una primera iteración, la configuración del Proxy se gestionaba mediante manipulación directa de archivos, sin coordinación entre *requests* concurrentes. Esta aproximación demostró ser problemática cuando múltiples dispositivos o procesos intentaban modificar configuración simultáneamente, resultando en estados inconsistentes y comportamiento impredecible del Proxy. La introducción de una base de datos SQLite como mediador para todas las operaciones de configuración resolvió este problema, aprovechando las propiedades transaccionales de la misma [19] para proteger contra condiciones de carrera.

Consideraciones de seguridad informaron múltiples aspectos del diseño arquitectónico. El Proxy maneja tráfico potencialmente sensible y debe garantizar confidencialidad e integridad de las comunicaciones. El soporte completo de TLS/SSL con terminación en el Proxy garantiza que el tráfico en tránsito esté protegido mediante cifrado. Las APIs de configuración del Proxy y del Mocking Engine están protegidas mediante autenticación y autorización, verificando que solamente usuarios autorizados puedan modificar comportamiento del sistema. La integración con el sistema de autenticación organizacional proporciona gestión centralizada de identidades y políticas de acceso, evitando la proliferación de credenciales y simplificando compliance con políticas de seguridad corporativas.

El contexto operacional de Mimick, diseñado específicamente para ambientes de desarrollo y testing en lugar de producción, permitió calibrar apropiadamente el nivel de inversión en controles de seguridad. Mientras que un sistema expuesto a internet o manejando datos de producción requeriría controles de seguridad exhaustivos, la operación de Mimick dentro de perímetros de red controlados y con datos sintéticos permite enfoque pragmático que equilibra seguridad con facilidad de uso. Esta calibración basada en contexto constituye práctica recomendada en ingeniería de seguridad, evitando tanto subinversión que dejaría el sistema vulnerable, como sobreinversión que agregaría complejidad sin beneficio proporcional.

Un riesgo identificado pero no completamente mitigado en la arquitectura actual concierne el versionado de Specs. El sistema no implementa actualmente versionado explícito de Specs, lo que significa que cuando un usuario modifica una Spec compartida, el cambio impacta inmediatamente a todos los usuarios que referencian esa Spec. Este comportamiento puede generar situaciones donde un usuario inadvertidamente rompe la funcionalidad de otros usuarios al modificar Specs en las que dependen. La estrategia de mitigación implementada se basa en el concepto de Scenarios, que actúan como templates o snapshots de configuraciones que pueden ser replicados entre usuarios. Un usuario puede clonar un Scenario existente antes de modificarlo, trabajando sobre su copia privada sin afectar a otros. Esta aproximación proporciona aislamiento parcial aunque no resuelve completamente el problema de versionado, área identificada como candidata para mejoras futuras del sistema.

La resiliencia del sistema ante fallos de componentes individuales es otro de los riesgos arquitectónicos. La naturaleza del deployment descentralizado del Proxy implica que el fallo de una instancia del Proxy afecta únicamente a su cliente o ambiente específico, sin impacto en otros. No obstante, para el cliente afectado, la caída del Proxy bloquea completamente la capacidad de comunicarse con otros servicios. Estrategias de mitigación incluyen monitoreo de salud del Proxy y alertamiento automático ante fallos, facilitando respuesta rápida. La arquitectura contenerizada del Proxy simplifica la recuperación mediante el reinicio automático de contenedores, aunque esto no protege contra fallos que requieran intervención manual. La evaluación de riesgo consideró implementar redundancia del Proxy mediante despliegue de múltiples instancias con balanceo de carga, pero el análisis de costo-beneficio determinó que la complejidad adicional no se justificaba para un sistema de desarrollo y testing donde downtime ocasional es tolerable.

La persistencia en PostgreSQL introduce dependencia en la disponibilidad de la base de datos. Un fallo de la base de datos impacta tanto la API Mimick como la Proxy API, bloqueando operaciones de gestión aunque no necesariamente afectando el procesamiento de tráfico ya configurado en el Proxy. Estrategias estándar de gestión de bases de datos como backups regulares, punto de recuperación definido, y potencialmente replicación para ambientes críticos, mitigan riesgo de pérdida de datos. La documentación operacional del sistema incluye procedimientos de recuperación ante fallo de base de datos, especificando pasos para restaurar desde backup y validar integridad de datos post-recuperación.

La evolución arquitectónica futura deberá considerar estos riesgos residuales y evaluar si cambios en el contexto de uso o en los requisitos del sistema justifican inversiones adicionales en mitigación. El balance actual entre complejidad, costo de desarrollo y mantenimiento, y cobertura de riesgos se considera apropiado para el estado actual del proyecto y su contexto operacional, aunque revisiones periódicas garantizarán que este balance se mantenga adecuado conforme el sistema evolucione y potencialmente expanda su alcance a casos de uso más críticos.

Este capítulo ha documentado la arquitectura técnica de Mimick, describiendo los componentes que materializan la propuesta conceptual presentada en el capítulo cuatro. El diseño resultante equilibra consideraciones de rendimiento, flexibilidad, mantenibilidad y gestión de riesgos, adoptando tecnologías maduras y patrones arquitectónicos probados. El siguiente capítulo explorará el proceso de desarrollo e implementación que transformó este diseño arquitectónico en un sistema funcional, incluyendo las fases de desarrollo, desafíos técnicos encontrados, y decisiones de implementación que refinaron la arquitectura durante la construcción del sistema.

6. Desarrollo e implementación

La arquitectura descrita en el capítulo anterior se materializó mediante un proceso de desarrollo iterativo que abarcó seis meses desde la concepción inicial hasta el deployment de una versión estable operando con usuarios reales. Este capítulo documenta las fases principales de este proceso, los aprendizajes obtenidos en cada etapa, y los desafíos técnicos más significativos encontrados durante la implementación, junto con las soluciones adoptadas para superarlos. La narrativa refleja una metodología pragmática de desarrollo que privilegió validación empírica temprana y ajustes continuos basados en feedback de usuarios, alineándose con principios contemporáneos de desarrollo ágil de software.

6.1 Fases de Desarrollo: PoC, MVP y Refinamiento Continuo

El desarrollo de Mimick siguió una progresión natural desde su validación conceptual hasta llegar a un sistema productivo, atravesando fases de complejidad creciente que permitieron validar suposiciones fundamentales antes de comprometer recursos en una implementación completa. Esta aproximación iterativa fue fundamental para identificar ajustes necesarios en el diseño arquitectónico y priorizar funcionalidades según su valor para usuarios finales.

6.1.1 Proof of Concept: Validación del modelo operacional

La fase inicial del proyecto se enfocó en validar que la propuesta de Mimick era viable dentro del modelo de desarrollo específico de la organización. El equipo decidió comenzar la implementación por la API de Mimicking, desarrollando primero la capacidad de definir Specs y servirlas en respuesta a *requests*. Esta decisión respondió a la intuición de que el core del valor estaba en la capacidad de gestionar y servir simulaciones, mientras que otros componentes como el Proxy se consideraban inicialmente accesorios.

Sin embargo, las primeras pruebas de concepto revelaron rápidamente una limitación fundamental: sin un componente de interceptación transparente, cada aplicación cliente requería modificaciones para apuntar directamente a la API de Mimicking, y no existía mecanismo para aislar sesiones sin requerir que cada aplicación gestionara explícitamente este aislamiento. Este descubrimiento motivó la incorporación del Proxy como componente arquitectónico central, modificando el diseño inicial.

El PoC incluyó una interfaz de usuario básica, aunque esta no constituía objetivo principal de validación en esta fase. La UI servía fundamentalmente como herramienta de testing interno, permitiendo al equipo de desarrollo interactuar con la API sin necesidad de construir *requests* HTTP manualmente. Esta decisión pragmática aceleró los ciclos de validación sin comprometer el foco en los aspectos arquitectónicos fundamentales.

La fase de POC se extendió por dos meses, culminando con una demostración funcional que validaba las suposiciones centrales del proyecto. El equipo identificó que los Scenarios, concepto que permitiría agrupar y reutilizar configuraciones de Specs entre usuarios, eran una funcionalidad clave para el valor de la herramienta. Adicionalmente, la necesidad de logging y telemetría emergió como requisito crítico: sin visibilidad sobre qué estaba sucediendo internamente en Mimick, el debugging y la comprensión del comportamiento del sistema resultaban excesivamente difíciles tanto para el equipo de desarrollo como para usuarios potenciales.

6.1.2 Minimum Viable Product: Primera versión funcional

Habiendo validado el concepto fundamental, el equipo procedió a construir un MVP que materializara las capacidades mínimas necesarias para entregar valor a usuarios reales. El alcance del MVP se definió cuidadosamente para equilibrar funcionalidad suficiente con tiempo de desarrollo

contenido, privilegiando características que habilitaran casos de uso completos aunque limitados en lugar de implementación parcial de funcionalidades ambiciosas.

El MVP incluyó cuatro componentes principales. Primero, la capacidad de definir Specs, aunque inicialmente sin interfaces de configuración mediante API, requiriendo definición directa en base de datos o mediante scripts. Esta limitación consciente permitió enfocar esfuerzo en garantizar que el motor de matching y servicio de respuestas funcionara correctamente antes de invertir en interfaces de gestión sofisticadas. Segundo, la API de Mocking completamente funcional, capaz de recibir *requests*, evaluar matches contra Specs configuradas, y retornar respuestas apropiadas. Tercero, un Proxy básico implementando aislamiento por sesión mediante evaluación del header *session_id*, redirigiendo tráfico a la API de Mocking o a servicios reales según configuración. Cuarto, una interfaz de usuario simplificada que permitía visualizar Specs existentes y configurar asignaciones básicas de Scenarios a sesiones, aunque con capacidades limitadas de edición.

El desarrollo del MVP requirió cuatro meses adicionales más allá del POC, totalizando seis meses desde el inicio del proyecto. Esta fase involucró no solamente implementación de funcionalidades sino también establecimiento de prácticas de desarrollo que garantizaran calidad de código y facilidad de mantenimiento futuro.

La validación del MVP se realizó con equipos de desarrollo internos de la organización, quienes representaban usuarios objetivo reales y podían proporcionar feedback fundamentado en casos de uso auténticos. Esta validación reveló áreas críticas de mejora. La experiencia de usuario de la interfaz web presentaba fricción significativa, requiriendo múltiples pasos y navegación no intuitiva para completar operaciones que los usuarios esperaban realizar rápidamente. El feedback confirmó además la importancia de Scenarios como mecanismo de compartición y reutilización de configuraciones, validando la intuición del equipo durante el POC. Finalmente, los usuarios expresaron necesidad de configurar Specs complejas directamente desde la UI, incluyendo reglas de matching sofisticadas y capacidades de templating en respuestas, funcionalidad que el MVP no proporcionaba forzando configuración manual en base de datos.

6.1.3 Iteraciones posteriores: Refinamiento continuo

Una vez validado el MVP con usuarios reales, el equipo adoptó un modelo de releases mensuales que incorporaba mejoras incrementales basadas en feedback continuo. Esta aproximación permitió evolucionar el producto de manera controlada, priorizando funcionalidades según su valor demostrado por usuarios activos en lugar de especulaciones sobre requisitos futuros. El proceso iterativo facilitó además la incorporación de aprendizajes operacionales: monitoreo de métricas de uso reveló patrones de comportamiento que informaron decisiones de producto, mientras que análisis de logs identificó casos borde que requerían manejo especial.

La metodología de desarrollo adoptada refleja una progresión natural desde la validación conceptual hasta el producto. El POC estableció viabilidad técnica y arquitectónica del enfoque propuesto, validando que la idea fundamental tenía sentido y era factible implementar. El MVP permitió validar en un entorno real con usuarios auténticos, confirmando que la solución generaba valor en contextos de uso verdaderos y revelando prioridades de funcionalidad que no habían sido evidentes en etapas conceptuales. Las iteraciones subsecuentes, guiadas por feedback de este uso real, refinaron la experiencia y expandieron capacidades en direcciones que respondían a necesidades demostradas en lugar de anticipadas.

6.1.4 Consideraciones sobre el enfoque inicial con inteligencia artificial

Durante las fases tempranas de conceptualización del proyecto, el equipo exploró un enfoque ambicioso que incorporaba inteligencia artificial como mecanismo central para gestionar estado persistente en las simulaciones. La propuesta arquitectónica contemplaba que cada request procesada por Mimick fuese interpretado por un modelo de IA que actuaría como motor de estado, modificando automáticamente una representación interna del sistema simulado para reflejar los efectos que la operación solicitada tendría en un sistema real. Por ejemplo, una request a un endpoint de transferencia bancaria no solamente retornaría una respuesta exitosa mockeada, sino que el modelo de IA actualizaría automáticamente los saldos de las cuentas involucradas en el estado interno. Subsecuentes *requests* que consultaran saldos de cuentas obtendrían valores consistentes con las transferencias previamente ejecutadas, creando la ilusión de interactuar con un sistema backend completamente funcional.

Esta visión resultaba particularmente atractiva para el contexto de aplicaciones bancarias, donde numerosos flujos implican secuencias complejas de operaciones con estado interdependiente. Un usuario autenticándose modifica su estado de sesión activa. Una transferencia afecta saldos de múltiples cuentas. Cambios en configuración de notificaciones persisten para subsecuentes consultas. La capacidad del modelo de IA de comprender semántica de operaciones y actualizar automáticamente estado correspondiente prometía eliminar necesidad de configuración manual exhaustiva de cada posible secuencia de interacciones, permitiendo que Mimick generalizara comportamiento apropiado incluso para flujos no explícitamente configurados.

La implementación de este enfoque motivó, de hecho, la arquitectura de estado por sesión descrita en capítulos anteriores. Cada *session_id* mantendría su propio estado independiente gestionado por el modelo de IA, permitiendo que múltiples desarrolladores ejecutaran secuencias de operaciones completamente diferentes sin interferencia. El modelo procesaría *requests*, inferiría modificaciones de estado necesarias, y las aplicaría al contexto de esa sesión específica, garantizando aislamiento completo entre usuarios mientras proporcionaba comportamiento stateful coherente dentro de cada sesión.

Sin embargo, la validación empírica de este enfoque durante desarrollo del POC y primeras iteraciones del MVP reveló limitaciones fundamentales que comprometían su viabilidad para una herramienta de desarrollo productiva. El rendimiento constituyó la primera barrera significativa: el procesamiento de cada request mediante el modelo de IA introducía latencias del orden de cientos de milisegundos a varios segundos, dependiendo de la complejidad de la inferencia requerida. Esta latencia resultaba inaceptable en un contexto donde los desarrolladores ejecutan frecuentemente secuencias rápidas de operaciones y esperan feedback inmediato. La percepción de lentitud generaba frustración y quebraba el flujo de trabajo natural de desarrollo.

Más crítico aún, la confiabilidad de las modificaciones de estado generadas por el modelo exhibía inconsistencia problemática. En numerosas ocasiones, el modelo cometía errores al inferir efectos de operaciones, modificando campos incorrectos del estado o aplicando transformaciones que violaban restricciones de integridad del dominio. Particularmente problemáticos eran casos donde el modelo "alucinaba" entidades o campos que no existían en la definición real de las APIs, generando estado internamente consistente desde la perspectiva del modelo pero incompatible con las expectativas de las aplicaciones cliente que esperaban estructuras de datos específicas. La corrección de estos errores requería intervención manual, eliminando las ventajas de automatización que motivaban el enfoque.

La complejidad de ciertos flujos de negocio excedía además las capacidades del modelo para capturar lógica correctamente. Operaciones bancarias involucran frecuentemente reglas complejas sobre límites de transacción, validaciones de disponibilidad, cálculos de comisiones que varían según tipo de cuenta y configuraciones del usuario, y estados transitorios durante procesamiento asíncrono. El modelo

de IA, entrenado sobre tráfico capturado, podía aprender patrones estadísticos pero no comprendía verdaderamente la semántica de negocio subyacente. Esto resultaba en comportamiento que superficialmente parecía razonable pero que bajo scrutinio revelaba inconsistencias con las reglas reales del dominio.

Finalmente, la naturaleza no determinística del modelo generaba experiencia de usuario insatisfactoria para una herramienta de desarrollo. La misma secuencia de requests ejecutada dos veces podía producir estados finales diferentes, dependiendo de variabilidad estocástica en la inferencia del modelo. Esta impredecibilidad comprometía la reproducibilidad de errores y dificultaba la confianza en Mimick para testing regresivo, donde la consistencia absoluta de comportamiento constituye un requisito fundamental.

La decisión de adoptar el enfoque pragmático descrito en capítulos anteriores respondió directamente a estos hallazgos empíricos. El equipo mantuvo la arquitectura de estado por sesión, reconociendo su valor fundamental para aislamiento entre usuarios, pero reemplazó la gestión automática mediante IA por mecanismos determinísticos basados en configuración explícita y merge de overrides. La captura de tráfico y generación de Specs mediante reglas programáticas proporcionó automatización suficiente para reducir esfuerzo de configuración, mientras que el control explícito sobre estado y respuestas garantizó predictibilidad y rendimiento aceptables.

Esta experiencia evidencia las tensiones entre la automatización basada en IA y las necesidades propias de las herramientas de desarrollo, donde determinismo, rendimiento y transparencia suelen ser prioritarios. El capítulo ocho desarrollará con mayor profundidad esta transición tecnológica y sus implicaciones más amplias para la evaluación de cuándo incorporar técnicas de inteligencia artificial en sistemas de software.

6.2 Desafíos técnicos y soluciones implementadas

El proceso de implementación de Mimick presentó múltiples desafíos técnicos que requirieron investigación, experimentación, y en algunos casos rediseño de componentes para arribar a soluciones satisfactorias. Esta sección documenta tres desafíos significativos encontrados, las soluciones implementadas, y los aprendizajes generalizables obtenidos de cada experiencia.

6.2.1 Modelado de estado para sobreescritura de respuestas

Uno de los requisitos fundamentales de Mimick es permitir que cada sesión de usuario pueda sobreescribir cualquier aspecto de la response generada por una Spec, adaptándola a sus necesidades específicas sin afectar a otros usuarios. Esta capacidad resulta crítica en numerosos casos de uso reales. Por ejemplo, en flujos de autenticación, la respuesta de login típicamente incluye un campo *device_id* que debe coincidir exactamente con el identificador del dispositivo móvil del usuario. Si este valor no coincide, la aplicación cliente rechaza la response y el usuario no puede acceder. Cada dispositivo de testing requiere por lo tanto su propia versión personalizada de esta response, modificando únicamente el campo *device_id* mientras mantiene el resto de la estructura intacta.

Comportamientos similares se manifiestan en múltiples servicios tanto JSON como XML a través de la aplicación bancaria. Algunos servicios retornan identificadores que deben ser consistentes con estado mantenido localmente en el dispositivo. Otros incluyen timestamps que deben estar dentro de ventanas temporales específicas para ser aceptados. La capacidad de sobreescribir estos campos selectivamente sin requerir duplicación completa de Specs constituye un habilitador fundamental para que Mimick escale a equipos grandes con múltiples desarrolladores trabajando simultáneamente.

El desafío técnico radica en implementar un modelo de datos y lógica de merge que permita especificar overrides parciales de manera intuitiva y que funcione consistentemente tanto para JSON como para XML. El enfoque ingenuo de almacenar respuestas completas sobreescritas para cada sesión resulta inviable: genera duplicación masiva de datos mayormente idénticos, dificulta actualización de Specs base cuando cambios deben propagarse a todas las sesiones, y complica razonamiento sobre qué aspectos de una response están siendo sobreescritos versus heredados de la Spec base.

La solución implementada introduce un modelo de estado por sesión que almacena únicamente las modificaciones específicas que el usuario ha realizado respecto a la configuración base. Este estado se persiste como estructura JSON que especifica paths dentro de la response y los valores que deben ser sustituidos. Cuando una request activa una Spec que tiene overrides en el estado de la sesión correspondiente, el sistema calcula el merge entre la response definida en la Spec y las modificaciones del estado. Este cálculo de merge se implementa mediante algoritmos que operan sobre representaciones de árbol de las estructuras JSON o XML, permitiendo especificar sustituciones en cualquier nivel de anidamiento mediante notación de path estándar.

La implementación de merge presentó sus propios desafíos. Existen casos borde donde la semántica de merge es ambigua: si la Spec define un array y el estado especifica modificación de un elemento particular, ¿debe el merge preservar otros elementos del array original, o reemplazar el array completo? Si un path especificado en el estado no existe en la response base, ¿debe el merge crear la estructura necesaria, o considerar esto un error? El equipo adoptó políticas explícitas para estos casos, priorizando comportamiento predecible y mensajes de error claros cuando overrides no pueden ser aplicados según lo especificado.

El aprendizaje fundamental de este desafío es que merges complejos entre estructuras de datos arbitrarias requieren políticas claras sobre casos borde y comunicación efectiva de estas políticas a usuarios. La documentación de Mimick incluye una sección dedicada explicando la semántica de merge y proporcionando ejemplos de configuraciones válidas e inválidas. La interfaz de usuario implementa validación que alerta a usuarios cuando intentan configurar overrides que no se comportarán según esperado, reduciendo confusión y frustraciones derivadas de resultados no intuitivos de merge.

6.2.2 Ruteo dinámico en el Proxy

El Proxy constituye punto de entrada para todo el tráfico gestionado por Mimick, y debe ser capaz de tomar decisiones de ruteo sofisticadas basándose en características de cada request. Específicamente, el Proxy debe evaluar el header que contiene el *session_id*, consultar configuración persistida para determinar si esa sesión tiene habilitada redirección a Mimick para el servicio siendo solicitado, y enrutar la request apropiadamente a la API de Mocking o al servicio real. Esta decisión debe tomarse con latencia mínima dado que ocurre en el path crítico de cada request, y debe soportar actualización dinámica de reglas sin requerir reinicio completo del Proxy.

El desafío radica en que NGINX, elegido como base del Proxy por sus características de rendimiento, no proporciona nativamente capacidades de ruteo basado en lógica arbitraria que consulte fuentes de datos externas. La configuración estándar de NGINX opera mediante archivos de configuración estáticos que especifican reglas de enrutamiento fijas. Si bien NGINX soporta recarga sin downtime, depender de generación y recarga de archivos de configuración para cada cambio en reglas de ruteo introduciría latencias inaceptables y complejidad operacional significativa.

La solución adoptada aprovecha OpenResty, distribución extendida de NGINX que incorpora el motor de scripting LuaJIT directamente en el contexto de procesamiento de *requests*. OpenResty permite implementar lógica de ruteo arbitrariamente compleja mediante scripts Lua que se ejecutan durante el

procesamiento de cada request con overhead mínimo. El equipo implementó un sistema de ruteo dinámico donde scripts Lua consultan estructuras de datos en memoria que representan las reglas de redirección actuales, determinando el destino apropiado para cada request basándose en hostname, path, y *session_id*.

Las estructuras de datos de reglas se inicializan al arranque del Proxy leyendo desde base de datos, y se actualizan en respuesta a cambios en la configuración. Esta arquitectura proporciona balance óptimo entre rendimiento de evaluación de reglas durante el procesamiento de *requests*, que opera completamente en memoria sin I/O, y actualización de reglas, que puede tolerar latencia de segundos sin impactar experiencia de usuario dado que cambios de configuración son eventos relativamente infrecuentes.

El aprendizaje clave de este desafío es el poder y flexibilidad que OpenResty proporciona para implementar lógica de proxy personalizada. La capacidad de programación mediante Lua en el plano de datos permitió al equipo implementar comportamientos sofisticados que serían extremadamente difíciles o imposibles usando la configuración estática de NGINX. El impacto en latencia resultó negligible en práctica, con mediciones mostrando overhead de menos de 20 milisegundos. Esta capacidad de extensión mediante scripting facilitó además la adición de funcionalidades adicionales al Proxy en iteraciones posteriores, como logging detallado de requisitos e implementación de ruteo granular por path, agregando apenas decenas de líneas de código Lua.

6.2.3 Matching complejo de requests

La API de Mimicking debe determinar qué Spec aplicar para cada request entrante, evaluando potencialmente múltiples candidatas que podrían hacer match. El matching debe soportar reglas arbitrariamente complejas que consideren no solamente hostname, path y método HTTP, sino también contenido de headers y body de la request. Múltiples Specs pueden hacer match parcialmente con una request, requiriendo mecanismo de desambiguación que seleccione la más apropiada de manera consistente y predecible.

Un caso de uso que ilustra esta complejidad es un servicio SOAP utilizado para búsqueda de productos por tipo. El servicio expone un único endpoint que acepta *requests* con estructura XML específica. Para simular comportamientos diferenciados según el tipo de producto siendo buscado, Mimick necesita inspeccionar el contenido del body XML, extraer el tipo de producto de campos específicos dentro de la estructura, y seleccionar la Spec apropiada que retorna la response correspondiente a ese tipo. Este nivel de matching basado en contenido excede las capacidades de matching simple por URL o headers.

La solución implementada divide el proceso de matching en dos fases. La primera fase ejecuta matching estándar evaluando hostname, path y método HTTP, identificando todas las Specs que coinciden exactamente en estas dimensiones. Esta fase opera mediante índices en base de datos y puede ejecutarse eficientemente incluso con grandes cantidades de Specs configuradas. Si la primera fase identifica una única Spec candidata, el matching se considera resuelto y esa Spec se utiliza.

Si múltiples Specs hacen match en la primera fase, el sistema procede a la segunda fase donde ejecuta las reglas de matching complejas almacenadas en cada Spec candidata. Estas reglas se expresan como predicados que evalúan propiedades arbitrarias de la request: presencia o valor de headers específicos, matches de expresiones regulares contra contenido del body, evaluación de paths XPath en XMLs, o queries JSONPath en payloads JSON. Las reglas se evalúan secuencialmente hasta encontrar la primera que retorna match exitoso, o hasta agotar candidatas en cuyo caso la request se considera no matcheada.

Esta arquitectura en dos fases proporciona rendimiento óptimo para el caso común donde matching simple por hostname/path/método es suficiente, evitando el costo de evaluación de reglas complejas

cuando no es necesario. Simultáneamente, habilita expresividad completa para casos donde el matching sofisticado es requerido, aceptando la latencia adicional como trade-off explícito. Mediciones en producción confirman que la mayoría de *requests* resuelven en la primera fase con latencias promedio de 100 milisegundos, mientras que *requests* que requieren segunda fase típicamente agregan 20-50 milisegundos de latencia, duración que resulta tolerable para el contexto de uso de Mimick.

El aprendizaje fundamental es la importancia de diseñar sistemas que optimicen para el caso común mientras mantienen capacidad de manejar casos complejos cuando sea necesario. La tentación de implementar únicamente matching simple habría limitado severamente la aplicabilidad de Mimick a casos de uso reales. Inversamente, evaluar siempre reglas complejas para todas las *requests* habría introducido latencia inaceptable. La arquitectura en dos fases proporciona el balance apropiado, y resultó además naturalmente extensible a nuevos tipos de criterios de matching que fueron agregados en iteraciones posteriores sin requerir reestructuración fundamental del sistema

7. Despliegue y evaluación

Este capítulo documenta el proceso de puesta en producción de Mimick, describe la metodología empleada para evaluar su efectividad, presenta los resultados obtenidos tanto cuantitativos como cualitativos, y establece una comparación sistemática con herramientas existentes en el mercado. La evaluación se fundamenta en datos recolectados durante meses de uso por un equipo de desarrollo real, proporcionando evidencia concreta sobre el impacto de la herramienta en la productividad y calidad del proceso de desarrollo de software.

7.1 Puesta en producción

El despliegue de Mimick en entornos reales de desarrollo se realizó aproximadamente seis meses después del inicio del proyecto, coincidiendo con la finalización del MVP documentado en el capítulo 6.

La infraestructura de despliegue se estableció en la nube de AWS de la empresa, aprovechando servicios gestionados para reducir la complejidad operacional. La arquitectura de despliegue utilizó contenedores Docker orquestados mediante Docker Compose, facilitando la gestión de los cinco componentes del sistema: Proxy, Proxy API, API Mimicking, base de datos PostgreSQL, y *frontend* web. Esta decisión arquitectónica proporcionó beneficios significativos en términos de portabilidad, aislamiento de dependencias, y facilidad de actualización. El proceso de despliegue se automatizó mediante un *pipeline* de CI/CD que ejecuta tests, construye imágenes de contenedores, y despliega automáticamente en el ambiente de desarrollo tras cada commit exitoso en la rama principal del repositorio. Las migraciones de esquema de base de datos se gestionaron mediante Alembic, garantizando evolución controlada y versionada del modelo de datos conforme el sistema incorporaba nuevas funcionalidades.

El despliegue inicial se dirigió a un equipo de desarrollo de aproximadamente quince personas trabajando en aplicaciones bancarias móviles. Esta dimensión de equipo proporcionaba suficiente diversidad de casos de uso para validar la propuesta de valor de Mimick mientras mantenía la complejidad operacional en niveles manejables para identificar y resolver problemas emergentes rápidamente. La adopción no fue mandatoria desde el inicio, permitiendo que los desarrolladores evaluaran la herramienta e incorporaran su uso gradualmente a sus flujos de trabajo. Los primeros cuatro desarrolladores comenzaron utilizando Mimick de manera experimental durante el primer mes,

explorando sus capacidades y proporcionando feedback temprano sobre usabilidad y limitaciones. Este grupo inicial actuó efectivamente como *early adopters*, identificando casos de uso valiosos y socializando las capacidades de la herramienta con el resto del equipo.

Durante los siguientes tres meses, el uso de Mimick se expandió orgánicamente al equipo completo conforme los desarrolladores reconocían valor en escenarios específicos de su trabajo diario. Los mecanismos de observabilidad implementados en el sistema, documentados en el capítulo seis, facilitaron el monitoreo de patrones de uso y la identificación de oportunidades de mejora basadas en comportamiento real de usuarios. Al momento de redacción de esta tesis, un segundo equipo de desarrollo se encuentra en proceso de *onboarding*, validando la escalabilidad del despliegue y la capacidad de Mimick de soportar múltiples equipos trabajando en proyectos diferentes con requisitos de simulación diversos.

7.2 Testing y validación

La capacidad de facilitar testing efectivo constituyó uno de los objetivos centrales del diseño de Mimick, y la validación de este aspecto requirió evaluación tanto del testing habilitado por la herramienta como del testing del sistema Mimick mismo.

7.2.1 Testing facilitado por Mimick

El impacto más significativo de Mimick se observó en la habilitación de testing automatizado *end-to-end*, categoría de pruebas que previamente resultaba extremadamente difícil de implementar dado el contexto del proyecto. Antes de la introducción de Mimick, el equipo de desarrollo carecía completamente de tests automatizados para flujos completos de usuario, limitándose a validación manual de funcionalidad durante el desarrollo y pruebas de regresión ejecutadas por el equipo de *Quality Assurance*. Esta ausencia de automatización no se debía a falta de intención o conocimiento técnico, sino a barreras fundamentales impuestas por la arquitectura del sistema y la naturaleza de las dependencias externas.

Los servicios bancarios de los cuales dependía la aplicación móvil presentaban dos características que impedían testing automatizado efectivo: mutabilidad de estado y disponibilidad inconsistente. Considérese el escenario de testing de una transferencia bancaria: la ejecución exitosa de una transferencia modifica el saldo de las cuentas involucradas, alterando el estado del sistema de manera que una segunda ejecución del mismo test produciría resultados diferentes, o fallaría debido a saldo insuficiente. Reproducir el estado inicial requería intervención manual para reconfigurar cuentas, proceso que tomaba entre tres y cinco días en el caso promedio, considerando los tiempos de respuesta de equipos de operaciones bancarias y las restricciones de seguridad en ambientes de prueba. Adicionalmente, la caída periódica de servicios externos por mantenimiento, problemas de red, o incidentes operacionales introducía errores en cualquier test que dependiera de estos servicios, minando la confianza en la suite de tests y reduciendo su utilidad como herramienta de validación de regresión.

La introducción de Mimick eliminó estas barreras fundamentales mediante la provisión de respuestas determinísticas y controladas para servicios externos. El equipo de desarrollo implementó una nueva herramienta interna específicamente diseñada para automatizar testing *end-to-end* aprovechando las capacidades de Mimick. Esta herramienta graba interacciones de usuarios reales con dispositivos móviles durante ejecución manual de flujos, capturando tanto acciones del usuario como las comunicaciones entre la aplicación y servicios backend. Las grabaciones se convierten en tests reproducibles que pueden ejecutarse automáticamente contra dispositivos utilizando Mimick para simular respuestas de servicios

externos, garantizando que cada ejecución del test observa exactamente el mismo comportamiento de servicios, independientemente del estado real de sistemas externos.

Al momento de redacción de esta tesis, se han implementado más de ochenta test cases automatizados cubriendo los flujos críticos de negocio de la aplicación bancaria: compra de productos de inversión, transferencias entre cuentas, y procesos de otorgamiento de préstamos. Estos tests se ejecutan automáticamente cada noche, proporcionando validación continua de que cambios en el código de la aplicación no han introducido regresiones en funcionalidad crítica.

La capacidad de probar casos borde constituyó otro beneficio significativo observado durante la evaluación. Escenarios como clientes con más de diez tarjetas de crédito, saldos excepcionalmente altos, respuestas malformadas de servicios, o condiciones de error específicas, resultaban difíciles de reproducir en ambientes de prueba tradicionales dado que requerían configuración manual en múltiples sistemas. Con Mimick, estos escenarios se configuran directamente especificando las respuestas que la aplicación debe procesar, habilitando testing exhaustivo de casos excepcionales sin necesidad de manipular estado en servicios externos.

Un beneficio emergente e inicialmente no anticipado de Mimick fue su utilidad para identificar problemas de performance en la aplicación móvil. Durante una demostración de las capacidades de Mimick, se configuró un *Scenario* simulando un cliente con una cantidad inusualmente grande de productos financieros. La ejecución de este escenario reveló que la capa de *rendering* de la aplicación exhibía performance significativamente degradada. La investigaciónidentificó que el código de *rendering* intentaba dibujar todos los elementos simultáneamente en lugar de procesarlos parcialmente conforme necesario, problema que no había sido detectado previamente debido a que los ambientes de desarrollo no contenían datos de prueba con volúmenes tan grandes. La facilidad con la cual Mimick permite configurar respuestas de tamaño arbitrario lo convierte en una herramienta útil para testing de performance, caso de uso que el equipo está explorando actualmente de manera más sistemática. Las mediciones preliminares confirman que los tiempos de respuesta de Mimick son consistentemente inferiores y presentan menor variabilidad que los servicios reales, característica deseable para aislar problemas de performance de la aplicación, de fluctuaciones en latencia de red o carga en servicios externos.

La capacidad de testing independiente y paralelo entre desarrolladores representa probablemente el beneficio más fundamental proporcionado por Mimick, directamente abordando la motivación principal del proyecto documentada en el capítulo tres. Antes de Mimick, solo uno o dos desarrolladores podían efectivamente trabajar y testear funcionalidades que dependían de servicios externos simultáneamente, dado que compartían un único ambiente de pruebas con configuración global. Modificaciones realizadas por un desarrollador afectaban inmediatamente el comportamiento observado por otros, generando confusión, trabajo duplicado, y necesidad de coordinación explícita para evitar conflictos. Con Mimick, cada uno de los quince desarrolladores del equipo puede trabajar completamente en paralelo sin interferencias, dado que cada sesión de desarrollo mantiene configuración independiente de servicios simulados. Este aislamiento reduce la necesidad de coordinación y permite que testing que previamente requería serialización pueda ejecutarse concurrentemente, acelerando significativamente los ciclos de validación.

7.2.2 Testing del sistema Mimick

La calidad y confiabilidad del sistema Mimick mismo fue validada mediante una estrategia comprehensiva de testing automatizado implementada desde las fases tempranas del desarrollo. El código base de Mimick alcanzó aproximadamente ochenta por ciento de cobertura de tests. Los tests se categorizaron en dos niveles principales: tests unitarios que validan comportamiento de funciones y

módulos individuales en aislamiento, y tests de integración que verifican interacción correcta entre componentes del sistema.

Los tests unitarios, implementados utilizando el framework pytest y cubren lógica de negocio crítica como el algoritmo de matching de *requests* contra Specs, el cálculo de merge entre Specs y *overrides* especificados en el estado de la sesión, y la generación automática de Specs a partir de tráfico capturado. Estos tests proporcionan feedback rápido durante el desarrollo, ejecutándose en cuestión de segundos y permitiendo que desarrolladores validen cambios localmente antes de commit al repositorio.

Los tests de integración verifican flujos completos que involucran múltiples componentes del sistema interactuando entre sí y con la base de datos. Ejemplos representativos incluyen: creación de una Spec mediante la API, configuración de un Scenario que referencia esa Spec, activación de una Run asociada al Scenario, y validación de que cuando llega un *request*, la Spec apropiada es seleccionada y su respuesta retornada.

La suite completa de tests se ejecuta automáticamente en el pipeline de CI/CD ante cada *commit* al repositorio, bloqueando el *merge* de cambios que introducen fallos en tests existentes. Esta validación continua fue instrumental para mantener la estabilidad del sistema conforme se agregaban nuevas funcionalidades y se modificaba el código durante las iteraciones documentadas en el capítulo seis.

7.3 Metodología de evaluación

La evaluación de la efectividad de Mimick requirió un enfoque metodológico que combinara métricas cuantitativas con feedback cualitativo de usuarios reales. Los objetivos primarios de evaluación se establecieron como: reducción de tiempo de desarrollo gracias a la eliminación de bloqueos causados por caida de servicios, incremento en la cantidad de pruebas automatizadas implementadas, mejora en capacidad de testear casos borde y escenarios complejos en ambientes tradicionales, y satisfacción de usuarios con la herramienta y su integración en flujos de trabajo existentes. Estos objetivos reflejan tanto beneficios medibles objetivamente como aspectos subjetivos relacionados con experiencia de usuario y adopción.

La recolección de datos se estructuró mediante tres mecanismos: Primero, instrumentación del sistema Mimick mismo para capturar métricas de uso mediante logging comprehensivo de operaciones realizadas por usuarios: creación de Specs, configuración de Scenarios, activación de Sesiones, y procesamiento de *requests*. Estas métricas proporcionaron evidencia cuantitativa sobre patrones de adopción y utilización de funcionalidades. Segundo, análisis de cambios en prácticas de desarrollo del equipo observables mediante artifacts del proceso de software: cantidad de tests automatizados implementados y frecuencia de ejecución de suites de testing. Tercero, recolección de feedback cualitativo mediante instancias formales de encuestas y entrevistas con usuarios durante el periodo de evaluación, complementadas con feedback informal continuo recopilado mediante canales de comunicación del equipo.

El periodo de evaluación formal se extendió desde la adopción completa de Mimick por el equipo de desarrollo, aproximadamente tres meses después del deployment inicial, hasta el momento de redacción de esta tesis. Esta duración proporcionó tiempo suficiente para que usuarios internalizaran la herramienta en sus prácticas de trabajo y para que los patrones de uso se estabilizaran, eliminando efectos de novedad característicos en la adopción inicial. La recolección de métricas de uso del sistema se configuró como proceso continuo automatizado, mientras que las instancias de feedback cualitativo se espaciaron estratégicamente para captar la evolución de los usuarios conforme se familiarizaban con la herramienta.

7.4 Resultados obtenidos

La evaluación de Mimick en uso real produjo evidencia significativa de impacto positivo tanto en métricas cuantitativas de adopción y uso como en feedback cualitativo de usuarios sobre valor percibido y limitaciones observadas.

7.4.1 Métricas cuantitativas

Los datos recolectados mediante instrumentación del sistema demuestran adopción y uso sostenido de Mimick. El equipo de desarrollo adoptó la herramienta de manera estable, constituyendo la base de usuarios activos durante el periodo de evaluación. Actualmente, un segundo equipo se encuentra actualmente en proceso de *onboarding*, validando la escalabilidad de la herramienta y expandiendo el alcance de la validación a contextos de desarrollo adicionales con requisitos diferentes.

El contenido creado por usuarios en el sistema fue de: 105 Specs configuradas representando servicios simulados, 25 Scenarios agrupando configuraciones reutilizables para casos de uso específicos, y más de 1000 Sesiones registradas incluyendo tanto sesiones de desarrollo interactivo como ejecuciones automatizadas de testing. El volumen de tráfico procesado por Mimick alcanzó un promedio de quince mil *requests* por semana, confirmando que la herramienta se integró efectivamente en el flujo de trabajo diario del equipo y no permaneció como solución utilizada únicamente en circunstancias excepcionales.

El impacto más dramático se observó en testing automatizado. Previo a la introducción de Mimick, el equipo carecía completamente de tests automatizados end-to-end, limitándose a validación manual de funcionalidad. La habilitación de simulación determinística de servicios externos mediante Mimick facilitó la implementación de una herramienta interna de testing automatizado, resultando en la creación de más de ochenta casos de prueba que se ejecutan automáticamente cada noche. Este incremento desde cero hasta más de ochenta tests representa una transformación fundamental en las prácticas de *quality assurance* del equipo, proporcionando validación continua de funcionalidad crítica y reduciendo significativamente el riesgo de regresiones no detectadas.

La reducción en tiempo de desarrollo observada fue principalmente cualitativa, con desarrolladores reportando que podían probar cambios en la aplicación más rápidamente y detectar errores más temprano en el ciclo de desarrollo. Si bien no se tienen métricas cuantitativas antes de la introducción de Mimick que permitan medición rigurosa de mejora, las estimaciones de desarrolladores sugieren una reducción de uno a dos días en el tiempo requerido para desarrollar y validar features que dependen de servicios externos. Esta mejora se atribuye primariamente a la eliminación de esperas por disponibilidad de servicios y a la capacidad de configurar escenarios de prueba rápidamente sin coordinación con otros equipos.

7.4.2 Feedback cualitativo

Las encuestas y entrevistas con usuarios revelaron patrones consistentes de feedback positivo complementado con identificación de limitaciones y oportunidades de mejora. Los aspectos más valorados de Mimick se centraron en la autonomía proporcionada a desarrolladores individuales y la capacidad de probar escenarios previamente inaccesibles.

La configuración independiente por sesión emergió como el beneficio más frecuentemente mencionado por los usuarios. Desarrolladores expresaron apreciación significativa por la capacidad de cada persona de mantener su propia configuración de servicios simulados sin afectar el trabajo de colegas. Esta autonomía eliminó la necesidad de coordinación explícita para evitar conflictos y redujo la fricción

asociada con testing de funcionalidades que dependen de servicios compartidos. La posibilidad de experimentar libremente con diferentes configuraciones sin preocupación de que estas modificaciones impacten a otros miembros del equipo fue identificada como transformadora para la velocidad de iteración durante el desarrollo.

Un beneficio no anticipado fue la mejora en documentación de servicios consumidos. La necesidad de crear Specs que describan explícitamente el comportamiento de servicios externos resultó en documentación estructurada que previamente no existía de forma centralizada y mantenida. Desarrolladores reportaron que consultar Specs existentes proporcionaba información más accesible y actualizada sobre la estructura de *requests* y *responses* que la documentación oficial de APIs bancarias, la cual frecuentemente presentaba discrepancias con implementaciones reales o directamente no existía.

Sin embargo, el feedback también reveló limitaciones significativas y áreas de mejora. La curva de aprendizaje de Mimick fue identificada como más pronunciada de lo deseable, con usuarios reportando dificultad inicial para entender la totalidad de funcionalidades disponibles y cómo aprovecharlas efectivamente. La interfaz web, si bien funcional, no fue percibida como suficientemente intuitiva, requiriendo exploración y experimentación considerable antes de que los usuarios se sintieran capaces de usar el sistema.

Una observación particularmente relevante fue la tendencia de usuarios de recurrir a servicios reales en lugar de configurar nuevos servicios en Mimick cuando enfrentaban requisitos no cubiertos por Specs existentes. El esfuerzo percibido de configurar un nuevo servicio simulado superaba el costo de simplemente utilizar el servicio real cuando este se encontraba disponible, resultando en que Mimick primariamente se utilizaba en situaciones donde servicios externos presentaban problemas o cuando se requerían escenarios específicos imposibles de lograr con servicios reales. Los usuarios solo reconocían valor significativo de Mimick cuando algo se rompía o necesitaban replicar escenarios particularmente complejos, sugiriendo que la integración de la herramienta en el flujo de trabajo diario no alcanzó el nivel de naturalidad inicialmente esperado.

Las solicitudes de funcionalidades adicionales se concentraron en dos áreas principales. Primero, capacidad de configurar Specs con reglas de matching más complejas y mejor detección de errores de configuración que facilite identificación y corrección de problemas en especificaciones. Segundo, configuración granular del Proxy a nivel de URL específica en lugar de solamente *hostname*, permitiendo control más fino sobre qué *requests* se *mockean* versus cuáles se enrutan a servicios reales. Esta última solicitud refleja casos de uso donde usuarios desean *mockear* solo subconjuntos específicos de las APIs de un servicio mientras mantienen comportamiento real para otros *endpoints*.

7.5 Comparación con herramientas existentes

La evaluación de Mimick requiere posicionamiento respecto a alternativas disponibles en el mercado. El capítulo dos documentó análisis de herramientas de simulación de servicios existentes: WireMock, Beeceptor, y Postman Mock Server. Esta sección profundiza en comparación estructurada que contrasta fortalezas y limitaciones relativas de estas herramientas versus Mimick, identificando casos de uso donde cada opción resulta más apropiada.

7.5.1 Mimick versus WireMock

WireMock es una herramienta muy utilizada en la simulación de servicios HTTP por la comunidad Java. Las diferencias arquitectónicas principales son: Mimick implementa aislamiento por sesión

permitiendo configuraciones independientes por desarrollador, mientras WireMock mantiene configuración global compartida. Mimick integra proxy híbrido para tráfico mixto transparente, mientras WireMock requiere que los mocks se invoquen como servicios independientes.

WireMock se destaca por su madurez (14+ años), simplicidad de despliegue (un binario JAR), integración nativa con ecosistema Java (JUnit), y arquitectura ligera con mejor performance. Mimick se adaptó mejor al proyecto por tres requisitos específicos: equipo de quince desarrolladores requiriendo configuración independiente simultánea, necesidad de tráfico híbrido transparente sin modificar código de aplicación, y restricciones de compliance bancario favoreciendo ejecución on-premise.

WireMock es más apropiado para proyectos centrados en Java, equipos pequeños donde la configuración compartida no genera conflictos, y contextos que prioricen performance donde la complejidad de arquitectura distribuida no se justifica.

7.5.2 Mimick versus Beeceptor

Beeceptor es un servicio en la nube que elimina requisitos de instalación. Las diferencias principales con Mimick son: Mimick requiere el despliegue de cinco componentes en infraestructura controlada, mientras Beeceptor opera completamente en su propia infraestructura. Mimick implementa configuración granular por sesión a nivel de URL individual, mientras Beeceptor proporciona configuración por endpoint compartida.

Beeceptor se destaca por su inmediatez (mocks funcionando en segundos), interfaz optimizada para inspección de tráfico HTTP en tiempo real, y modelo *freemium* que elimina costos de mantenimiento de infraestructura. Mimick se adaptó mejor al proyecto por dos requisitos críticos: regulaciones bancarias prohibiendo que tráfico con datos sensibles transite por servidores externos, y necesidad de configuración independiente para cada uno de los desarrolladores que trabajan simultáneamente.

Beeceptor es más apropiado para prototipado rápido dónde el tiempo de configuracion es crítico, proyectos sin restricciones regulatorias sobre el uso de servicios cloud, y contextos donde la inspección visual de tráfico HTTP en tiempo real es muy importante.

7.5.3 Mimick versus Postman Mock Server

Las diferencias principales con Mimick son: Mimick implementa un proxy que intercepta tráfico transparentemente, mientras Postman Mock Server requiere de configuracion manual por servicio. Mimick mantiene configuración aislada por sesión, mientras Postman gestiona configuración mediante *workspaces* compartidos con potencial de conflictos. Mimick se diseñó para proyectos multi-servicio, mientras Postman se optimizó para *mocking* básico de contratos API.

Postman se destaca por su ecosistema integrado (Collections, Tests, Monitors, Documentation en plataforma única), adopción masiva que minimiza curva de aprendizaje, y capacidades de colaboración nativas con sincronización en tiempo real. Mimick se adaptó mejor al proyecto por requisitos específicos: necesidad de *mockear* selectivamente servicios sin modificar código de aplicación, configuración completamente independiente entre los distintos usuarios de la plataforma, y workflows bancarios complejos que podían ser guardados como Scenarios.

Postman Mock Server es más apropiado para equipos que ya usan extensivamente Postman, proyectos donde la documentación está integrada y donde el mocking de contratos se combina con testing automatizado.

7.5.4 Limitaciones de Mimick

Reconocer las limitaciones de Mimick respecto a las alternativas es esencial para la comprensión de su posicionamiento. Cuatro categorías principales de limitaciones emergen de la comparación.

Primero, ecosistema y extensibilidad: Mimick presenta arquitectura más monolítica sin sistema de plugins ni integraciones predefinidas con otras herramientas. WireMock proporciona un sistema de extensiones e integraciones con Spring Boot, JUnit, y TestContainers. Postman ofrece un ecosistema completo de herramientas complementarias. Beeceptor soporta webhooks e integraciones con herramientas de CI/CD. Esta menor flexibilidad reduce la capacidad de adaptar Mimick a flujos de trabajo específicos o extender funcionalidad sin modificar el código base del sistema.

Segundo, capacidades avanzadas de simulación: Mimick se enfoca en grabado y replay de tráfico más que simulación programática avanzada. WireMock soporta request matching sofisticado mediante JSONPath, XPath, regex, y custom matchers; generación dinámica de contenido a través de templates programables; y simulación configurable de delays y timeouts. Postman proporciona variables de entorno y scripts *pre-request/post-response* en JavaScript. Beeceptor ofrece templates para generación de datos aleatorios. Para escenarios que requieren generación dinámica compleja de datos o simulación de condiciones de red, estas herramientas ofrecen funcionalidad más rica predefinida.

Tercero, simplicidad y portabilidad: Mimick requiere infraestructura compleja con cinco componentes que deben ser desplegados, configurados, y mantenidos: Proxy, Proxy API, API Mimicking, base de datos PostgreSQL, y frontend web. WireMock se distribuye como un JAR ejecutable en cualquier entorno con JVM sin dependencias adicionales. Beeceptor es un servicio cloud que no requiere despliegue. Postman es una aplicación de escritorio o cloud sin infraestructura adicional. Esta complejidad introduce mayor fricción en la adopción, costos operacionales continuos, y dificultades de mantenimiento.

Cuarto, comunidad y madurez: Como proyecto interno, Mimick no posee una comunidad establecida y su documentación está en desarrollo continuo. WireMock cuenta con catorce años de historia y documentación exhaustiva. Postman tiene una comunidad masiva con recursos educativos extensos y soporte para grandes empresas.

7.5.5 Síntesis comparativa

La Tabla 2 consolida las comparaciones presentadas en las secciones anteriores, proporcionando una visión integrada de las fortalezas y limitaciones relativas de cada herramienta. Esta síntesis facilita la identificación de contextos donde cada solución resulta más apropiada, evitando generalizaciones sobre superioridad absoluta que no reflejan la realidad de necesidades heterogéneas en distintos proyectos de desarrollo.

Criterio	WireMock	Beeceptor	Postman	Mimick
Aislamiento por sesión	No	No	No	Sí
Proxy híbrido nativo	No	Básico	No	Sí
Generación automática de specs	No	Parcial	No	Sí
Ejecución on-premise	Sí	No	Parcial	Sí

Persistencia de escenarios	Limitada	No	Collections	Sí (por sesión)
Madurez del proyecto	14+ años	~5 años	10+ años	1 año
Ecosistema y plugins	Extenso	Webhooks	Completo	Ninguno
Simulación programática	JSONPath, regex, templates	Templates básicos	Scripts JS	Grabado/replay
Simplicidad de despliegue	1 JAR	Sin instalación	App desktop	5 componentes
Comunidad y documentación	Extensa	Moderada	Masiva	Interna
Licenciamiento	OSS / Comercial	Freemium	Freemium	Interno

Tabla 2. Síntesis comparativa de herramientas de simulación de servicios

La comparación sugiere que Mimick resulta más apropiado en contextos que cumplan tres condiciones: equipos de desarrollo medianos a grandes (más de 8 desarrolladores) donde múltiples desarrolladores requieren configuraciones independientes simultáneas, entornos con restricciones regulatorias que demanden ejecución on-premise y control sobre el tráfico de datos sensibles, y proyectos donde la velocidad de configuración inicial mediante grabado automático de tráfico resulte prioritaria sobre capacidades avanzadas de simulación programática.

Esta evaluación comparativa refuerza la conclusión de que la selección de herramientas de simulación de servicios debe fundamentarse en el análisis de requisitos específicos del contexto, evitando la adopción de soluciones basada únicamente en popularidad o características técnicas aisladas. La contribución de Mimick al panorama de herramientas disponibles radica precisamente en cubrir un nicho específico que las alternativas establecidas no abordan adecuadamente: el desarrollo colaborativo con aislamiento por sesión en entornos regulados.

8. Lecciones aprendidas

El desarrollo de Mimick proporcionó aprendizajes significativos tanto en dimensiones técnicas como en aspectos estratégicos de decisiones arquitectónicas y de negocio. Este capítulo sintetiza las lecciones más relevantes, organizadas en dos categorías principales: aprendizajes técnicos generalizables a otros proyectos de software, y reflexiones sobre el proceso de cambiar el enfoque inicial centrado en inteligencia artificial hacia una solución fundamentada en principios determinísticos. La documentación de estas lecciones pretende proporcionar valor más allá del contexto específico de Mimick, contribuyendo al cuerpo de conocimiento sobre desarrollo de herramientas para equipos de software y sobre aplicación apropiada de tecnologías emergentes como inteligencia artificial en herramientas de desarrollo.

8.1 Aprendizajes técnicos

El primer aprendizaje concierne la importancia crítica del aislamiento por sesión en herramientas de desarrollo colaborativo. La decisión arquitectónica de implementar configuración completamente

independiente para cada identificador de sesión se reveló como habilitador fundamental del valor de Mimick. Sin este aislamiento, el sistema habría replicado la limitación principal de herramientas existentes: configuraciones compartidas que generan bloqueos y conflictos entre desarrolladores trabajando simultáneamente. El aprendizaje generalizable es que herramientas diseñadas para uso por múltiples desarrolladores deben priorizar mecanismos que eliminen o minimicen la coordinación explícita requerida entre usuarios. La capacidad de trabajar independientemente, sin preocupación de que acciones propias afecten a colegas, reduce significativamente la fricción cognitiva y mejora la productividad del equipo. Este principio aplica no solamente a herramientas de mocking sino a categorías más amplias de software de desarrollo: ambientes de testing, sistemas de gestión de datos de prueba, herramientas de debugging distribuido, entre otros.

El segundo aprendizaje se relaciona con el poder y flexibilidad de OpenResty para implementar lógica de proxy personalizada. La evaluación inicial de NGINX como base del Proxy consideró primariamente sus características de rendimiento y confiabilidad. La capacidad de programación mediante Lua emergió durante la implementación como ventaja diferencial más significativa de lo anticipado. Scripts Lua relativamente concisos permitieron implementar comportamientos sofisticados de ruteo, logging y transformación de tráfico que habrían sido extremadamente difíciles o imposibles usando la configuración estática tradicional de NGINX. El costo de performance introducido por ejecutar estos scripts resultó negligible en práctica, contradiciendo preocupaciones iniciales de que estos añadirían latencia inaceptable. El aprendizaje generalizable es que plataformas que combinan rendimiento de componentes compilados de bajo nivel con capacidades de extensión mediante scripting embebido ofrecen balance óptimo para casos de uso que requieren tanto performance como flexibilidad. Este patrón se observa en diversos contextos: sistemas de bases de datos con procedimientos almacenados, herramientas de red con hooks programables, aplicaciones con plugins basados en lenguajes de scripting. La resistencia a adoptar estas capacidades de extensión por preocupaciones de performance resulta injustificada si la implementación subyacente es eficiente, como demuestra nuestra experiencia con OpenResty donde la latencia medida fue inferior a veinte milisegundos incluso para lógica de ruteo compleja.

El tercer aprendizaje concierne trade-offs entre automatización y control explícito en herramientas de desarrollo. La experiencia con el enfoque basado en inteligencia artificial documentada más adelante en este capítulo ilustra que los desarrolladores valoran predictibilidad y transparencia por sobre brevedad en configuración cuando estas características entran en conflicto. Herramientas que automatizan aspectos de configuración o comportamiento deben garantizar que esta automatización sea confiable, transparente en su funcionamiento, y que pueda ser sobreescrita cuando los usuarios necesitan control fino. La tentación de utilizar técnicas sofisticadas de IA o machine learning como "caja negra" debe balancearse contra la necesidad de los desarrolladores que requieren entender exactamente qué está sucediendo en sus herramientas, particularmente cuando analizan problemas o validan comportamiento. Este principio no implica que la automatización sea indeseable, sino que la automatización basada en reglas determinísticas y heurísticas explícitas suele proporcionar un balance superior a aproximaciones que sacrifican transparencia por utilizar una IA como mecanismo “caja negra”.

Adicionalmente, el proyecto confirmó el valor fundamental de observabilidad y logging en herramientas de desarrollo. La implementación de RequestLogs, inicialmente considerada funcionalidad secundaria, emergió como componente crítico que facilitaba la investigación de problemas tanto para usuarios de Mimick como para el equipo de desarrollo del sistema mismo. La capacidad de inspeccionar exactamente qué había ocurrido durante el procesamiento de *requests* elimina la ambigüedad y acelera la resolución de problemas. Este aprendizaje refuerza el principio general de que las herramientas de desarrollo deben proveer observabilidad en lugar de operar como "cajas negras", proporcionando

visibilidad comprehensiva sobre su operación interna. La inversión en instrumentación y logging detallado se justifica no solamente para debugging durante el desarrollo del sistema, sino como funcionalidad permanente que proporciona valor continuo a usuarios durante la operación normal.

8.2 Reflexiones sobre la transición del enfoque basado en IA hacia un enfoque determinístico

El proceso de desarrollo de Mimick incluyó una fase inicial donde el equipo exploró el uso de inteligencia artificial generativa como componente central de la arquitectura del sistema. Esta exploración, que se extendió aproximadamente seis meses antes de la decisión de moverse hacia el enfoque de mocking documentado en esta tesis, proporcionó lecciones valiosas sobre la aplicación de tecnologías de IA en herramientas de desarrollo.

8.2.1 Motivación inicial del enfoque basado en IA

La decisión de explorar inteligencia artificial como componente de Mimick se fundamentó en observaciones iniciales que sugerían viabilidad técnica del enfoque. Experimentos preliminares demostraron que los modelos de lenguaje grandes (Large Language Models) exhibían capacidad para generar respuestas estructuradas que respeten contratos específicos. Dados prompts apropiados que describan la estructura esperada de las respuestas JSON o XML, los modelos producían contenido sintácticamente correcto y semánticamente razonable. Esta capacidad sugería potencial para resolver uno de los problemas más complejos en simulación de servicios con estado: la modificación del estado dentro de sistemas complejos.

Un ejemplo de esto es la simulación de un servicio bancario. Una operación de transferencia debe no solamente retornar una respuesta indicando éxito, sino también debe modificar el saldo de las cuentas origen y destino de manera que los próximos *requests* reflejen el nuevo estado. La implementación de esta lógica de negocio mediante reglas explícitas requiere código específico para cada tipo de operación y cada modificación de estado que esta produce. La visión del enfoque basado en IA era que un modelo suficientemente capaz podría inferir las modificaciones de estado apropiadas basándose en descripciones de alto nivel de la operación realizada, eliminando la necesidad de programación explícita de lógica de negocio y permitiendo que Mimick genere respuestas que respeten este contrato.

8.2.2 Exploración técnica y modelos evaluados

El equipo exploró múltiples proveedores de modelos de lenguaje, evaluando tanto servicios cloud como la posibilidad de usar modelos en infraestructura propia. Las pruebas de concepto iniciales se enfocaron en escenarios simplificados diseñados para validar la viabilidad fundamental del enfoque: servicios con ocho o diez operaciones, estado consistiendo en un puñado de campos numéricos o strings, y lógica de negocio relativamente simple expresable en lenguaje natural. En estos contextos limitados, los modelos demostraron capacidad razonable para generar modificaciones de estado coherentes y respuestas apropiadas.

La implementación de nuestro prototipo representaba el estado del sistema como estructura JSON que se incluía en el prompt enviado al modelo junto con una descripción de la operación siendo ejecutada. El modelo retornaba tanto la respuesta que debía enviarse al cliente como las modificaciones que debían aplicarse al estado para reflejar los efectos de la operación. Esta implementación permitía al sistema

mantener consistencia de estado a través de múltiples operaciones sin requerir programación explícita de reglas de transición de estado.

8.2.3 Problemas técnicos identificados

Conforme la exploración progresaba hacia casos de uso más realistas y complejos, emergieron problemas técnicos fundamentales que cuestionaban la viabilidad de esta solución. Estos problemas se categorizaron en tres áreas principales.

El primer problema era el tiempo de respuesta de los modelos. La invocación de un LLM mediante API introduce latencia significativa, que puede ir de segundos a minutos en el peor de los casos, dependiendo del tamaño del contexto y la complejidad de la generación requerida. Para Mimick, que trataba de emular *requests* de aplicaciones que esperan respuestas con latencias comparables a servicios HTTP normales (cientos de milisegundos), esta latencia resultaba inaceptable. Pruebas con dispositivos móviles reales revelaron que la lentitud introducida por el procesamiento basado en IA hacía la interacción con la aplicación perceptiblemente más lenta, degradando la experiencia de usuario de manera que los desarrolladores probablemente optarían por no utilizar nuestra herramienta incluso cuando técnicamente solucionara sus problemas de simulación de servicios.

El segundo problema se relacionaba con el crecimiento del tamaño de representación del estado interno. Para los modelos de lenguaje, el contexto proporcionado en cada invocación cuenta con límites de ventana de contexto que restringen la cantidad de información que puede ser procesada. El estado de aplicaciones bancarias reales es mucho más complejo que los casos de ejemplo utilizados en nuestra validación inicial: clientes con múltiples cuentas, cada cuenta con historial de transacciones, productos financieros con configuraciones elaboradas, relaciones entre entidades que deben mantenerse consistentes. La representación textual de este estado crecía rápidamente hacia dimensiones que excedían capacidades de modelos disponibles, o que consumían fracciones significativas de la ventana de contexto dejando poco espacio para instrucciones y generación de output. Estrategias de optimización como incluir únicamente subconjuntos relevantes del estado o comprimir representaciones introducían complejidad adicional y riesgo de que el modelo careciera de información necesaria para generar modificaciones de estado correctas.

El tercer problema, y potencialmente el más fundamental, era la complejidad de lógica de negocio en aplicaciones reales. Los casos de ejemplo utilizados en validación inicial involucraban reglas relativamente simples: transferencias que suman y restan montos, verificaciones de saldo suficiente, actualizaciones de fechas. La lógica de negocio bancaria real incluye reglas sustancialmente más complejas: cálculo de intereses compuestos, comisiones dependientes de múltiples factores, validaciones que consideran regulaciones específicas, flujos de aprobación multi-paso, actualizaciones en cascada que propagan cambios a través de múltiples entidades relacionadas. Lograr que un modelo de lenguaje respetara fielmente estas reglas requeriría descripciones extremadamente detalladas y precisas, aproximándose a programación declarativa en lenguaje natural, cuestionando si el enfoque realmente proporcionaba ventajas sobre implementación directa mediante código.

8.2.4 Problemas funcionales y predictibilidad

Más allá de los desafíos técnicos, el equipo identificó una categoría fundamental de problemas relacionados con la naturaleza probabilística de los modelos de lenguaje y las expectativas de predictibilidad en herramientas de desarrollo. Incluso asumiendo que los problemas técnicos de latencia y tamaño de contexto pudieran resolverse mediante optimizaciones o modelos más potentes, permanecía la

cuestión de si utilizar IA en el camino crítico de una herramienta de desarrollo constituía una decisión apropiada para la experiencia de usuario.

Las herramientas de desarrollo operan en contextos donde predictibilidad y determinismo son altamente valorados [14]. Cuando un desarrollador configura un comportamiento esperado y ejecuta un test, anticipa que el resultado será exactamente el especificado, permitiendo razonamiento preciso sobre esa funcionalidad. La introducción de un componente de IA que procesa lenguaje natural y genera outputs mediante procesos probabilísticos introduce incertidumbre fundamental. No existe garantía absoluta de que el modelo producirá exactamente el mismo resultado dado un mismo estímulo, especialmente cuando este estímulo está sujeto a interpretación o cuando el contexto cambia sutilmente.

El objetivo central de Mimick era reducir la incertidumbre en el proceso de desarrollo eliminando dependencias de servicios externos impredecibles y permitiendo que desarrolladores trabajaran en ambientes controlados y reproducibles. La paradoja del enfoque basado en IA era que introducía una nueva fuente de incertidumbre y potenciales retrasos: comportamiento no determinístico del modelo, necesidad de análisis cuando el modelo no generaba el resultado esperado, y dependencia de servicios externos (APIs de proveedores de LLMs) que presentaban sus propias características de disponibilidad y latencia. En lugar de eliminar problemas, este enfoque trasladaba la dependencia de servicios bancarios externos a dependencia de servicios de IA externos, sin resolver fundamentalmente la motivación principal del proyecto.

8.2.5 Proceso de decisión y momento del cambio de enfoque

El reconocimiento de estos problemas no fue instantáneo sino gradual, emergiendo conforme las pruebas progresaban desde casos simplificados hacia escenarios productivos. El momento más significativo en el proceso de decisión ocurrió durante las pruebas con dispositivos móviles reales ejecutando la aplicación bancaria completa. La lentitud perceptible introducida por el procesamiento de IA hacía la interacción frustrante, confirmando que la latencia no era meramente una métrica abstracta sino que impactaba directamente en la experiencia de usuario. Este momento de validación cristalizó preocupaciones que habían estado emergiendo durante las semanas de exploración técnica.

Otro momento crítico ocurrió cuando el equipo analizó la complejidad de las reglas de negocio bancarias que debían simularse. La inspección de documentación de APIs y el análisis de comportamiento observado en servicios reales revelaron lógica mucho más elaborada que los casos de ejemplo utilizados en prototipos. La proyección del esfuerzo requerido para describir estas reglas en lenguaje natural con suficiente precisión para que un modelo las respetara sugería que nuestro enfoque no simplificaba significativamente la implementación versus un enfoque programático tradicional, cuestionando la propuesta de valor fundamental de la aproximación basada en IA.

La decisión de ca fue gradual más que abrupta, emergiendo del consenso del equipo conforme acumulábamos evidencia de limitaciones técnicas y funcionales. La transición hacia el enfoque deterministico documentado en esta tesis no representó un abandono en la innovación sino el reconocimiento de que la innovación apropiada para Mimick no residía en la aplicación de técnicas de IA para la generación de respuestas, sino en el diseño de una arquitectura con aislamiento por sesión, el proxy híbrido transparente, y la captura estructurada de tráfico que facilita la configuración sin introducir incertidumbre o complejidad excesiva.

8.2.6 Lecciones generalizables sobre aplicación de IA

La experiencia del cambio de enfoque en torno al uso de IA en Mimick ofrece lecciones generalizables sobre la aplicación de inteligencia artificial en sistemas de software. La primera lección es la de identificar correctamente los contextos donde aplicar Inteligencia Artificial: en herramientas de desarrollo donde la predictibilidad es fundamental, la IA funciona mejor como asistencia opcional que mejora la experiencia sin comprometer la funcionalidad principal, mientras que en aplicaciones donde cierto grado de incertidumbre es aceptable (sistemas de recomendación, generación de contenido), las técnicas probabilísticas pueden ocupar un rol más central.

La segunda lección refiere a la importancia de la validación temprana con usuarios reales: métricas aisladas de latencia o precisión pueden ser engañosas, el impacto real en la experiencia de usuario solo emerge cuando se evalúan prototipos funcionales en condiciones representativas. Esta validación debe priorizarse dado que decisiones arquitectónicas fundamentales son extremadamente costosas de revertir una vez que el desarrollo ha progresado lo suficiente.

Finalmente, la lección más general es que la innovación apropiada en herramientas de desarrollo no necesariamente reside en la aplicación de técnicas de última generación, sino en el diseño cuidadoso de arquitecturas que eliminan la fricción y permiten workflows eficientes. El valor diferencial de Mimick proviene de decisiones arquitectónicas como el aislamiento por sesión, proxy híbrido y captura estructurada de tráfico. Estas decisiones no requieren técnicas avanzadas de *machine learning* pero proporcionan beneficios concretos que resuelven problemas reales de los usuarios. La tentación de incorporar tecnologías emergentes debe balancearse contra la evaluación de si realmente abordan las necesidades fundamentales del problema, o si su complejidad adicional distrae de soluciones más directas y efectivas.

9. Conclusiones y trabajo futuro

Este capítulo final sintetiza las contribuciones principales de Mimick al campo de virtualización de servicios, evalúa el cumplimiento de los objetivos establecidos al inicio del proyecto, identifica las limitaciones actuales del sistema, y propone direcciones concretas para el trabajo futuro tanto a nivel técnico como estratégico. La reflexión presentada aquí se fundamenta en la experiencia documentada a lo largo de los capítulos anteriores, proporcionando una evaluación de logros y oportunidades de mejora que puede informar el desarrollo de herramientas similares en otros contextos.

9.1 Síntesis de resultados

El problema central que motivó el desarrollo de Mimick era la dependencia crítica de servicios externos que bloqueaba el progreso de equipos grandes de desarrollo cuando estos servicios experimentaban caídas o no proporcionaban los datos necesarios para trabajar. Esta dependencia se manifestaba en bloqueos operacionales que impactaban directamente la productividad, imposibilitaban el testing exhaustivo de casos borde, y generaban conflictos entre desarrolladores que necesitaban simular diferentes estados del mismo servicio simultáneamente.

La respuesta a este problema se materializó en una arquitectura fundamentada en el aislamiento por sesión mediante identificadores únicos que permiten aplicar configuraciones independientes a cada usuario o flujo de trabajo. Este concepto, aunque simple en su formulación, aborda una limitación fundamental de herramientas existentes donde la configuración global compartida genera conflictos inevitables en entornos colaborativos. La arquitectura que permite que distintos usuarios obtengan

respuestas diferentes al mismo estímulo, con redirección de tráfico gestionada por sesión, constituye un patrón generalizable aplicable más allá del contexto específico de Mimick y representa la contribución conceptual más significativa del proyecto al campo de virtualización de servicios.

Esta innovación arquitectónica se validó no solo técnicamente sino en su adopción orgánica por parte del equipo de desarrollo. Más allá de las métricas cuantitativas documentadas en el capítulo siete, el logro más significativo de Mimick fue su incorporación genuina al trabajo diario del equipo. La transición desde una herramienta experimental hacia un componente estable de la infraestructura de desarrollo valida que el aislamiento por sesión no era meramente una solución elegante en lo teórico, sino que abordaba necesidades reales. La herramienta logró insertarse en los flujos de trabajo establecidos sin generar fricción excesiva. La arquitectura propuesta habilitó escenarios que resultan complejos o imposibles con herramientas tradicionales: la ejecución simultánea de pruebas con configuraciones mutuamente incompatibles, la reproducción de incidentes de producción sin interferir con el trabajo de otros miembros del equipo, y el desarrollo paralelo verdaderamente independiente donde cada desarrollador puede configurar su propio entorno de simulación sin coordinación explícita.

La evidencia de esta adopción sostenida, confirma que el enfoque de aislamiento por sesión no solo es técnicamente factible sino que proporciona valor tangible en contextos de desarrollo colaborativo a escala. Esta adopción representa un indicador de éxito más significativo que cualquier métrica aislada de uso, dado que refleja que Mimick resolvió problemas reales.

9.2 Cumplimiento de objetivos

La evaluación del cumplimiento de objetivos requiere contrastar los criterios específicos establecidos en el capítulo 3.4 con los resultados obtenidos tras el despliegue y adopción de Mimick. El proyecto definió tres objetivos principales con criterios cuantificables que permiten una evaluación objetiva del éxito alcanzado.

- Desarrollo paralelo sin conflictos

Los criterios establecidos para este objetivo incluían soportar al menos 15 desarrolladores simultáneos, mantener latencias por debajo de 250ms, y procesar al menos 100 requests por segundo.

El cumplimiento de este objetivo fue exitoso en todos sus criterios. La arquitectura de aislamiento por sesión permitió que el equipo completo de 15 desarrolladores utilizara Mimick simultáneamente sin experimentar interferencias entre configuraciones, como se documenta en los patrones de uso del capítulo 7. Las mediciones de rendimiento confirmaron latencias consistentes entre 150-200ms en operación normal, cumpliendo holgadamente el criterio de sub-250ms y representando una mejora de 10x sobre las latencias de 1.5-2 segundos observadas con servicios externos reales. El sistema demostró capacidad para procesar más de 15,000 requests semanales, lo que se traduce en picos sostenidos superiores a los 100 requests por segundo requeridos durante las ventanas de mayor actividad de desarrollo y testing.

- Reproducción ágil de incidentes de producción

El criterio principal establecía que cualquier incidente de producción debía poder reproducirse en un entorno controlado dentro del mismo día laboral.

Este objetivo se cumplió satisfactoriamente. La combinación del componente de Proxy que captura tráfico real y la capacidad de Mimick para servir respuestas grabadas previamente permitió consistentemente reproducir incidentes el mismo día de su reporte. El capítulo 7 documenta casos

concretos donde incidentes que anteriormente requerían entre 2 y 10 días para configurar manualmente los escenarios necesarios, ahora se reproducen en cuestión de horas. La funcionalidad de captura automática de tráfico HTTP y generación de especificaciones eliminó la complejidad técnica del proceso, permitiendo que cualquier miembro del equipo, incluyendo QA sin conocimiento profundo de la herramienta, pudiera capturar y reproducir comportamientos problemáticos.

- Viabilización de testing automatizado integral

El criterio establecido aspiraba a alcanzar al menos 50% de cobertura en pruebas end-to-end para casos críticos del negocio, partiendo de una situación donde la ejecución confiable de estas pruebas era prácticamente imposible.

Este objetivo presenta cumplimiento parcial pero significativo. Si bien el proyecto habilitó la creación de más de 80 casos de prueba automatizados (partiendo de cero), la cobertura efectiva de casos críticos se estima en aproximadamente 35-40%, por debajo del 50% objetivo. La brecha se debe principalmente a dos factores identificados:

Primero, la limitación técnica de Mimick respecto al contenido dinámico en las respuestas. Las Specs actuales solo soportan respuestas estáticas, impidiendo simular servicios que requieren valores calculados dinámicamente como timestamps actuales o tokens con expiración. Esta carencia afecta aproximadamente al 15% de los casos de prueba críticos que involucran validaciones temporales o secuencias con estado mutable.

Segundo, limitaciones en el ecosistema de testing móvil que trascienden el alcance de Mimick. Ciertos escenarios críticos requieren simular componentes nativos del dispositivo móvil (cámara, biometría, componentes criptográficos), elementos fuera del dominio de virtualización de servicios HTTP. Estos casos representan otro 10-15% de la cobertura objetivo.

A pesar de no alcanzar completamente el criterio del 50%, el avance desde cero casos automatizados hasta más de 80 representa un cambio fundamental en las capacidades de testing del equipo. Las pruebas ejecutan de manera determinística, eliminando los falsos negativos que plagaban intentos anteriores de automatización. Para operaciones administrativas del proxy, las latencias se mantuvieron consistentemente bajo el segundo establecido como aceptable, no impactando el flujo de pruebas.

- Evaluación global

El proyecto cumplió completamente dos de sus tres objetivos principales y logró avance sustancial en el tercero. Los criterios de rendimiento (latencia <250ms, 100 req/s) se superaron consistentemente. La capacidad de reproducir incidentes el mismo día transformó fundamentalmente el proceso de diagnóstico. El testing automatizado, aunque no alcanzó la meta del 50% de cobertura, estableció una base sólida con más de 80 casos automatizados donde antes no existía ninguno.

Estos resultados validan que Mimick abordó efectivamente los problemas centrales que motivaron su desarrollo, aunque queda espacio para mejoras incrementales, particularmente en el soporte para contenido dinámico que cerraría la brecha hacia el objetivo completo de cobertura de testing.

9.3 Limitaciones y trabajo futuro

La evaluación crítica de Mimick revela limitaciones que informan tanto su evolución futura como el desarrollo de herramientas similares. En la dimensión funcional, el sistema no implementa soporte robusto para ciertos patrones de acceso típicos de APIs REST como paginación, filtros complejos, o

recursos anidados. Desde la perspectiva de usabilidad, la curva de aprendizaje es moderada pero mejorable, principalmente debido a la ausencia de documentación completa y ejemplos exhaustivos que faciliten la adopción por nuevos usuarios. Los usuarios solicitan frecuentemente capacidades de emulación de latencias y timeouts configurables, así como soporte para templates programáticos en respuestas. Adicionalmente, dos áreas de deuda técnica requieren atención: la gestión de sesiones, que si bien cumple funcionalmente, podría optimizarse, y el almacenamiento de RequestLogs, que sin estrategia de rotación o borrado puede crecer indefinidamente representando un riesgo operacional.

El roadmap inmediato contempla el desarrollo de pantallas de configuración específicas para el Proxy, implementación de un modelo de usuarios y permisos para operación segura en entornos multi-equipo, creación de documentación comprensiva, y la incorporación de telemetría mediante OpenTelemetry para monitorear performance y uso del sistema.

Mirando más hacia adelante, nuestro plan es expandir la aplicabilidad de Mimick mediante integraciones estratégicas con el ecosistema de herramientas de desarrollo de la organización. Las integraciones con pipelines de CI/CD permitirían ejecutar tests automatizados como parte del proceso de build, abordando parcialmente la limitación identificada respecto a testing automatizado y garantizando que cambios en el código no introduzcan regresiones. Esta integración con CI/CD constituye un paso natural hacia la consolidación de Mimick como componente fundamental del flujo de desarrollo, donde la virtualización de servicios no sería una actividad separada sino parte integral del ciclo de desarrollo y validación continua. La expansión a otros equipos dentro de la organización que enfrentan desafíos similares con dependencias de servicios externos validará la generalidad del enfoque más allá del caso de uso original. De particular interés es el uso emergente de Mimick para demostraciones a potenciales clientes en escenarios donde servicios externos reales no están disponibles, expandiendo su propuesta de valor más allá de desarrollo y testing.

Esta evolución conecta directamente con la visión a largo plazo de consolidar Mimick como herramienta interna de uso amplio dentro de la organización. La decisión estratégica de mantener Mimick como solución interna, sin aspiraciones de generalización externa o comercialización, responde tanto a consideraciones técnicas como de negocio. La implementación actual está atada a herramientas internas de despliegue, y las funcionalidades planificadas de usuarios y permisos utilizarán sistemas de autenticación propietarios, incrementando el acoplamiento con la infraestructura interna.

9.4 Reflexiones finales

La retrospectiva sobre el desarrollo completo de Mimick proporciona lecciones que informan no solamente la evolución futura de esta herramienta específica sino también el enfoque hacia proyectos similares. La principal reflexión concierne la decisión de explorar inteligencia artificial como componente central de la arquitectura durante los primeros seis meses del proyecto. Con el conocimiento actual, esta inversión de tiempo en el enfoque basado en IA debería haberse evitado o al menos acortado significativamente, aprovechando ese periodo para desarrollar e iterar sobre el enfoque determinístico que eventualmente se adoptó, o para explorar aplicaciones más acotadas de IA donde la propuesta de valor fuera más clara y los riesgos más controlables.

Sin embargo, consideramos que varias decisiones fueron acertadas. Plantear el Proxy como componente separado y modular desde el inicio proporcionó flexibilidad arquitectónica fundamental que facilitó la iteración y mejora de ese componente sin afectar al resto del sistema. Esta modularidad permitió, por ejemplo, la evolución desde una implementación básica de proxy hacia la solución basada en OpenResty documentada en el capítulo seis, sin requerir cambios en la API de Mimicking o en el

frontend web. La decisión de incorporar telemetría simple desde el inicio para monitorear uso y performance del sistema fue igualmente valiosa, proporcionando visibilidad que informó decisiones de priorización y facilitó diagnóstico de problemas reportados por usuarios. Si bien la telemetría actual es básica y será expandida mediante OpenTelemetry como parte del roadmap documentado en este capítulo, la presencia de instrumentación desde etapas tempranas estableció una cultura de observabilidad que impactó en el desarrollo del sistema.

La experiencia documentada en esta tesis demuestra que el desarrollo de herramientas efectivas para equipos de software requiere un balance cuidadoso entre ambición técnica y pragmatismo operacional. La tentación de aplicar técnicas sofisticadas como inteligencia artificial o de construir arquitecturas comprehensivas que anticipen todo requisito futuro potencial debe balancearse con el enfoque en resolver problemas inmediatos de manera simple y validar suposiciones con usuarios reales temprano y frecuentemente. Mimick logró proporcionar valor tangible a equipos de desarrollo no por la sofisticación de sus componentes individuales sino por el diseño cuidadoso de una arquitectura que eliminó fricción existente y proporcionó capacidades específicas, como aislamiento por sesión, que abordaban limitaciones reales de las herramientas disponibles. Esta lección sobre la importancia de la simplicidad enfocada por sobre la completitud técnica constituye quizás la contribución más generalizable de este proyecto de desarrollo.

Bibliografia

[1] Farahmandpour, Z., Seyedmahmoudian, M., & Stojcevski, A. (2021). A review on the service virtualisation and its structural pillars. *Applied Sciences*, *11*(5), 2381. https://doi.org/10.3390/app11052381

[2] Newman, S. (2021). *Building microservices*. “O’Reilly Media, Inc.”

[3] Tornhill, A., & Borg, M. (2022, March 8). *Code Red: The Business Impact of code Quality -- A quantitative study of 39 proprietary production codebases*. arXiv.org. https://arxiv.org/abs/2203.04374

[4] Wiremock. (n.d.). *GitHub - wiremock/wiremock: A tool for mocking HTTP services*. GitHub. https://github.com/wiremock/wiremock

[5] Postman. (n.d). *Postman: The world's leading API platform* https://www.postman.com/

[6] Beeceptor. (n.d.). *Beeceptor - Rest & SOAP API Mock Server*. https://beeceptor.com/

[7] Fielding, R. T. (2000). *Architectural styles and the design of network-based software architectures*.

[8] Upadhyay, S., Mukherjee, H., & Acharya, A. A. (2015). Continuous Testing of Service-Oriented Applications Using Service Virtualization. IOSR Journal of Computer Engineering, 17(2), 88-92.

[9] Hine, C., Schneider, J., Han, J., & Versteeg, S. (2016). Enterprise Software service emulation: Constructing Large-Scale testbeds. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1605.06729

[10] *Beeceptor Pricing - Comes with Free, Individual, Team & Enterprise tiers*. (n.d.). API Mock Server. https://beeceptor.com/pricing/

[11] Dingsøyr, T., & Lassenius, C. (2016). Emerging themes in agile software development: Introduction to the special section on continuous value delivery. *Information and Software Technology*, *77*, 56–60. https://doi.org/10.1016/j.infsof.2016.04.018

[12] Kevrekidis, K., Albers, S., Sonnemans, P. J. M., & Stollman, G. M. (2009). Software complexity and testing effectiveness: An empirical study. *Proceedings, Annual Reliability and Maintainability Symposium/Proceedings. Annual Reliability and Maintainability Symposium*, *16*, 539–543. https://doi.org/10.1109/rams.2009.4914733

[13] Versteeg, S., Du, M., Schneider, J., Grundy, J., Han, J., & Goyal, M. (2016). Opaque Service Virtualisation: a practical tool for emulating endpoint systems. *arXiv (Cornell University)*. https://doi.org/10.48550/arxiv.1605.06670

[14] Mårtensson, T., Ancher, G., & Ståhl, D. (2023). Test environments for large‐scale software systems—An industrial study of intrinsic and extrinsic success factors. *Software Testing Verification and Reliability*, *33*(3). https://doi.org/10.1002/stvr.1839

[15] Jabbari, R., Ali, N. B., Petersen, K., & Tanveer, B. (2016). What is DevOps? *Association for Computing Machinery*, 1–11. https://doi.org/10.1145/2962695.2962707

[16] *Frequently asked questions*. (n.d.). WireMock. https://wiremock.org/docs/faq/#how-to-manage-many-mocks-across-different-use-cases-and-teams

[17] *NGINX Benchmark Benchmark* - OpenBenchmarking.org. (n.d.). https://openbenchmarking.org/test/pts/nginx

[18] *OpenResty® - Open source*. (n.d.). https://openresty.org/en/

[19] *SQLite is transactional*. (n.d.). https://sqlite.org/transactional.html