

Testing

Playbook para desarrolladores de software

Transformación desarrollo software - #ONE

Control de versiones

Versión	Fecha de Creación	Responsable	Descripción
1.0	{28-4-2023}	GSD	Primera versión oficial de los playbooks

Índice

1. Introducción	5
1.1. Acerca de este playbook	5
1.2. Principios básicos	5
1.3. Niveles de exigencia	6
2. Prácticas	7
2.1. Colaborar con el negocio para definir la funcionalidad a probar y las condiciones de funcionamiento del sistema con el objetivo de mitigar el riesgo	8
2.1.1. Colaborar con el negocio para definir la funcionalidad a probar y las condiciones de funcionamiento del sistema con el objetivo de mitigar el riesgo	9
2.2. Responsabilizar al equipo de la calidad del software que produce	13
2.2.1. Responsabilizar al equipo de la calidad del software que produce	14
2.3. Automatizar lo que se pueda para probar de forma eficiente, evidenciable, y detectar errores pronto	17
2.3.1. Automatizar las pruebas que lo requieran	17
2.3.2. Integrar pruebas en los pipelines de construcción y despliegue	21
2.3.3. Priorizar la ejecución de pruebas con el mayor impacto en cada fase	23
2.4. Seguir las mejores prácticas para cada tipo de prueba	26
2.4.1. Análisis estático	27
2.4.2. Pruebas unitarias	28
2.4.3. Pruebas de integración	30
2.4.4. Contract Tests	31
2.4.5. Pruebas de UI	33
2.4.6. Pruebas end to end	33
2.4.7. Pruebas de aceptación	35
2.4.8. Pruebas de seguridad	36
2.4.9. Pruebas de rendimiento	38
2.4.10. Pruebas de accesibilidad	39
2.4.11. Pruebas de monitorización sintética	40
2.5. Mantener los entornos compartidos estables	43
2.5.1. Mantener los entornos compartidos estables	44
2.6. Medir resultados e impulsar acciones de mejora de forma continua	46
2.6.1. Medir resultados e impulsar acciones de mejora de forma continua	47
3. Proceso	49
Anexo	51

Tipos de pruebas	51
Habilitadores	54
Entornos y dispositivos	55
Entornos locales o aislados	56
Entornos efímeros	58
Entornos compartidos persistentes	59
Datos de prueba	60
Matriz precondiciones	62
Herramientas	67

1. Introducción

1.1. Acerca de este playbook

El presente playbook (parte de una serie de seis) es un elemento fundamental de las prácticas de desarrollo de software del Banco. En él se definen los principios, la forma de organización, el proceso, las prácticas y las herramientas que todos los equipos de desarrollo de software del Banco deben adoptar en el proceso de evaluar y verificar que los productos de software cumplen los requisitos esperados. La excelencia en la realización de pruebas es un factor fundamental para acortar el plazo de comercialización, ya que permite detectar antes los defectos, reforzar la fiabilidad de las aplicaciones y mejorar la postura de ciberseguridad.

El playbook debe ser actualizado regularmente basándose en las prácticas y el panorama tecnológico del Banco. El equipo de GSD (Global Software Development) será el responsable de actualizarlo.

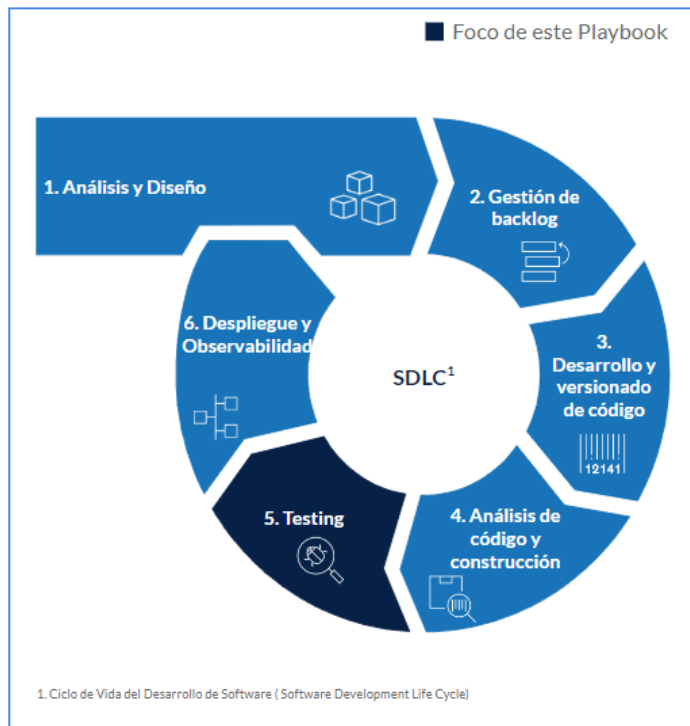


Figura 1 - Ciclo de vida del desarrollo de software

1.2. Principios básicos

Este playbook se guía por un conjunto de principios que todos los desarrolladores deben seguir:

- **Estandarización:** Usar una taxonomía global y formas de trabajo alineadas entre los equipos y geografías permitirá a los desarrolladores colaborar mejor, compartir mejor el conocimiento y rotar entre los equipos.

- **Calidad sobre cantidad:** Escribir código de calidad debe ser nuestra prioridad ya que la mayor parte de nuestro tiempo lo empleamos manteniendo software heredado y abordando deuda técnica, cumplir con esto mejorará la velocidad de desarrollo de los equipos a medio y largo plazo.
- **Transparencia y trazabilidad:** El total control y visibilidad de los sistemas y el código del Banco mejora la experiencia de los desarrolladores para navegar en los distintos entornos, responder a incidentes y garantizar el cumplimiento de los requisitos regulatorios.
- **Propiedad:** La alta demanda de calidad requerida en los servicios críticos del banco implica un control detallado de responsabilidad y propiedad de los activos de código. “Debemos ser propietarios del código que desarrollamos y sentirnos orgullosos de ello”.
- **Pequeños incrementos frecuentes frente a grandes cambios:** Grandes cantidades de código pueden poner en riesgo la estabilidad de nuestros sistemas ya que son más difíciles de probar y deshacer. Además, la mayoría de los sistemas de software evolucionan en función de los comentarios de los usuarios finales y los requisitos son difíciles de anticipar de antemano. Por tanto, es importante desarrollar rápidamente pequeños incrementos para aprender rápidamente y rectificar.
- **Simplicidad:** Los sistemas de software rara vez se escriben una vez y no se modifican. Más bien, muchas personas trabajan con el mismo sistema a lo largo del tiempo. Por eso es importante buscar las soluciones más sencillas que puedan comprenderse fácilmente y evolucionar en el futuro.
- **Automatización:** Los procesos manuales que son repetitivos deben automatizarse para mejorar nuestra productividad, enfocarnos en el desarrollo y reducir los errores por intervención manual.
- **Proporcionalidad:** La variedad y exhaustividad del testing a ejecutar estará en consonancia con la criticidad y taxonomía de la aplicación objeto de prueba.

1.3. Niveles de exigencia

Este playbook utiliza intencionadamente las siguientes tres palabras para indicar los niveles de exigencia según la norma [RFC2119](#):

- **Debe** - Significa que la práctica es un requisito absoluto, salvo las excepciones que se autoricen de forma específica.
- **Debería** - Significa que pueden existir razones válidas en circunstancias particulares para ignorar una práctica, pero deben comprenderse todas las implicaciones y sopesar cuidadosamente antes de elegir un camino diferente.
- **Podría** - Significa que una práctica es realmente opcional.

2. Prácticas

Los equipos de software del Banco deben **fomentar una cultura de testing** y considerar las pruebas como un entregable fundamental del proyecto desde el principio, incluso en los MVPs, integrándolas como parte del código, y comprender el valor de la inversión en la automatización de pruebas para ganar, entre otras cosas, confiabilidad en los sistemas.

Los miembros del equipo deben ser capaces de **aplicar distintos tipos de pruebas para detectar defectos lo antes posible** en el ciclo de desarrollo de software (SDLC) con inversiones justificables en el desarrollo y mantenimiento del código de prueba y las pruebas manuales.

En este capítulo se describen seis **prácticas** que todos los equipos del Banco deben seguir y sus respectivos beneficios, con independencia de la tecnología y el lenguaje de programación utilizados. Este puede ser un punto de partida para crear una cultura robusta de pruebas e ingeniería en el Banco y garantizar la calidad de los sistemas y aplicaciones:

- [1. Colaborar con el negocio para definir la funcionalidad a probar y las condiciones de funcionamiento del sistema con el objetivo de mitigar el riesgo](#)
- [2. Responsabilizar al equipo de la calidad del software que produce](#)
- [3. Automatizar lo que se pueda para probar de forma eficiente, evidenciable, y detectar los errores pronto](#)
- [4. Seguir las mejores prácticas para cada tipo de prueba](#)
- [5. Mantener los entornos compartidos estables](#)
- [6. Medir resultados e impulsar acciones de mejora de forma continua](#)

A continuación se definen cada una de las prácticas siguiendo la siguiente **estructura**:

- Sus beneficios (que conseguimos).
- Precondiciones (condiciones para poder aplicar la práctica).
- Adopción (cómo implementarla).
- Herramientas necesarias.
- Indicadores para los desarrolladores (como la medimos).
- Enlaces de interés.

Para ayudar a la lectura de este capítulo se han incluido las siguientes secciones en el **anexo** del documento:

- [Tipos de pruebas](#) definiendo los tipos de prueba comentados en las distintas prácticas.
- [Habilitadores](#) más relevantes para poder implementar las prácticas descritas.
- [Herramientas](#) globales necesarias para implementar las prácticas descritas.

2.1. Colaborar con el negocio para definir la funcionalidad a probar y las condiciones de funcionamiento del sistema con el objetivo de mitigar el riesgo

Beneficios

Los beneficios principales de promover la colaboración entre negocio y los equipos de desarrollo en las pruebas son:

- Determinar el alcance de una manera colaborativa y así poder establecer las funcionalidades más críticas para poder optimizar el esfuerzo.
- Tener una comprensión clara y compartida de cómo debe comportarse el software en diferentes situaciones y dónde hacer foco en las pruebas, lo que mejora la comunicación, la colaboración y la calidad del software entregado al cliente o usuario final.
- Promover el funcionamiento correcto de las operaciones críticas y la mejora de los puntos propensos a fallos.
- Mayor conciencia de las pruebas desde el principio, lo que conduce a una mejora en la calidad.
- Detección temprana de defectos, lo que reduce el costo, esfuerzo y tiempo requeridos para corregirlos.
- Aumentar la frecuencia de despliegue manteniendo o mejorando la calidad gracias a la automatización.
- Cumplimiento de las normas y regulaciones bancarias.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita por grado de aplicabilidad, alineado con las directrices de la práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Trazabilidad tests-defectos	Build automatizada Pruebas bajo demanda Trazabilidad e2e Convenciones Repositorios funcionales Código-pruebas juntos Soporte pruebas Soporte IDE Entornos y pruebas locales Pruebas pre-deploy Enforcement merge Pruebas post-deploy / compartidos Carga de datos periódicas	Pruebas configurables Pruebas planificadas Versionado SUT Pruebas impacto InnerSource repos InnerSource pipelines Pruebas pre-deploy Efímeros/asilados Pruebas pre-deploy virtualizadas Saltar controles seguridad Enforcement despliegues Versionado test-app-entornos

	Buscador de datos Carga de datos bajo demanda	Cambios entre pruebas Canaries/Feature Toggles Interceptación sistemas externos Trazas sistemas Monitorización y alertas
--	--	--

Adopción

2.1.1. Colaborar con el negocio para definir la funcionalidad a probar y las condiciones de funcionamiento del sistema con el objetivo de mitigar el riesgo

Los service owners (junto con negocio) **deben** marcar cuáles de todos los **comportamientos** de su sistema son **críticos** (e.g., alineados a criterios bancarios como los de continuidad de negocio) o **propensos a fallos** y por lo tanto requieren una automatización y monitorización continua.

Ayudando a los desarrolladores a decidir dónde centrar el esfuerzo de automatización de pruebas y definir el equilibrio correcto entre riesgo, coste y beneficios desde el inicio.

El responsable de producto, de forma conjunta con el equipo de desarrollo:

- **Debe** asumir las **pruebas desde el inicio** y definir **de forma colaborativa** los comportamientos a implementar y sus criterios de aceptación, centrándose en el comportamiento del software y en cómo éste debe interactuar con los usuarios y el sistema e incluyendo también escenarios alternativos o de excepción que sean relevantes para alguna de las partes (negocio, ingeniería, operaciones, etc). Asegurando así que los objetivos del proyecto y los requisitos del software se comprenden claramente entre todos los stakeholders y se cumplen a lo largo del tiempo gracias a las pruebas automáticas.
- **Debe** definir los comportamientos no esperados (a.k.a. unhappy path) desde un punto de vista funcional (e.g., hacer transferencias cuando no existen fondos).
- **Debe** centrarse en el funcionamiento desde el punto de vista de los usuarios e identificar **escenarios o casuísticas importantes** a considerar y probar.

El responsable de producto, de forma conjunta con el equipo de desarrollo:

- **Debe** ponderar el uso de **pruebas manuales** cómo **pruebas exploratorias, beta y advanced testing** (p. ej., pruebas de intrusión y rendimiento) para complementar las pruebas programadas dónde el contexto lo justifique.
- **Debe** preocuparse y definir **requisitos no funcionales** importantes a considerar, por ejemplo, la carga máxima prevista y los criterios de rendimiento exigidos, e implementar

pruebas de carga en consecuencia; p. ej., la primera imagen de una IU debería cargarse en menos de 100 ms con 100 conexiones concurrentes.

- **Debe** identificar los componentes externos impactados por los cambios en su software para definir un conjunto de pruebas de impacto adecuadas.

Cada requisito funcional:

- **Debe** estar almacenado en una herramienta de gestión de backlog global (e.g., Jira). Revisar el [playbook de backlog management](#) para más información.
- **Debe** tener un identificador universal, que deberá estar referenciado en aquellos tests que contribuyan a validarlo.

Los equipos **deberían** automatizar el mayor número posible de pruebas siempre que el tiempo y el coste necesario sea rentable. Además, deben buscar los casos más óptimos (e.g., automatizar la ejecución de pruebas E2E más críticas, automatizar la ejecución de tests unitarios optimizando el tiempo de los mismos).

Los siguientes tipos de código **deben** estar cubiertos por pruebas automatizadas (salvo excepciones justificadas):

- **Funcionalidad crítica**, que tiene graves consecuencias económicas, reputacionales, en el servicio, o impactos similares si falla, por ejemplo, operaciones con mercados, amortizaciones, transferencias, componentes altamente utilizados como los procesos de autenticación, etc.
- **Funcionalidad compleja**, con lógica difícil de entender o casos de validación no evidentes dónde es importante garantizar su funcionamiento con futuros cambios, por ejemplo asegurar que las penalizaciones se apliquen bien cuando los pagos se hacen tras la fecha límite, asegurar el funcionamiento de los límites diarios de transacciones, etc.
- **Código propenso a errores**, el código que ha demostrado fallar más a menudo que otras partes del código, por ejemplo, los hotfixes de incidentes de producción **deben** ser cubiertos con pruebas para evitar futuros incidentes.
- **Pruebas repetitivas**, por ejemplo, las pruebas manuales de regresión o las que se ejecutan a menudo, por ejemplo, con cada commit, merge o release. Los equipos **deberían** realizar un seguimiento de la ejecución de las pruebas manuales para identificar las pruebas repetitivas. En este punto es importante la colaboración de aquellos equipos con una visión más transversal y puedan identificar estos casos.

Los equipos **deberían** seguir un enfoque [ATDD](#) con requisitos de [desarrollo basado en el comportamiento \(BDD\)](#) y **pueden** usar documentos en forma de declaraciones

«Given-When-Then» utilizando [Gherkin](#) o similares para los comportamientos que se hayan priorizado para automatización por su criticidad.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jira** (o herramienta similar si no es posible su uso): Servicio global para la gestión de proyectos.
- **Activity Report** (o herramienta similar): Dónde analizar el esfuerzo de pruebas manuales.

Indicadores

Los indicadores que apliquen, deben ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Automatización de los criterios de aceptación	Porcentaje de nueva funcionalidad cuyo criterio de aceptación se valida mediante pruebas automáticas; y categorizadas por criticidad (e.g alta, media, baja).
Operativas críticas con pruebas automatizadas	Porcentaje de operativas críticas (e.g saldos, transferencias, etc) del servicio con pruebas automatizadas y cuyo criterio de aceptación se ha acordado con negocio.
Funcionalidad con criterios de aceptación	Porcentaje de funcionalidad con criterios de aceptación definidos

Enlaces

- <https://martinfowler.com/bliki/TestDrivenDevelopment.html>
- https://en.wikipedia.org/wiki/Acceptance_test-driven_development
- https://es.wikipedia.org/wiki/Desarrollo_guiado_por_comportamiento
- [https://en.wikipedia.org/wiki/Cucumber_\(software\)](https://en.wikipedia.org/wiki/Cucumber_(software))

2.2. Responsabilizar al equipo de la calidad del software que produce

Beneficios

Los beneficios clave de promover la responsabilidad de los equipos en la calidad del software que producen son:

- Evitar que los costes de desarrollo aumenten drásticamente debido a una mala calidad del software.
- Detectar errores de manera más eficiente al considerar las pruebas desde el inicio del proyecto.
- Ampliar la visión de las pruebas al poder recoger escenarios que hubieran pasado desapercibidos por no conocer ciertos detalles de implementación.
- Optimizar los procesos mediante la participación de los equipos de Desarrollo en las pruebas y en su automatización.
- Incrementar la frecuencia de entregas sin comprometer la calidad y manteniendo los costes gracias al uso de la automatización en todas las fases del proceso.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita por grado de aplicabilidad, alineado con las directrices de la práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Trazabilidad tests-defectos Monitorización y alertas	Build automatizada Pruebas configurables Pruebas bajo demanda Pruebas planificadas Trazabilidad e2e Convenciones Repositorios funcionales Código-pruebas juntos Soporte pruebas Soporte IDE Entornos y pruebas locales Pruebas pre-deploy Enforcement merge Pruebas post-deploy / compartidos Trazas sistemas Carga de datos periódicas Buscador de datos Carga de datos bajo demanda	Versionado SUT Pruebas impacto InnerSource repos InnerSource pipelines Pruebas pre-deploy efímeros/asilados Pruebas pre-deploy virtualizadas Saltar controles seguridad Rollback automático Enforcement despliegues Versionado test-app-entornos Cambios entre pruebas Canaries/Feature Toggles Interceptación sistemas externos Reserva de datos Usuarios productivos de pruebas

Adopción

2.2.1. Responsabilizar al equipo de la calidad del software que produce

Los responsables del servicio, del producto y los equipos de desarrollo **deben** ser responsables de la calidad del software que producen. Los equipos **deberían** promover el [Dogfooding](#) para experimentar en primera persona cómo funcionan los productos y detectar cualquier problema o error antes de que los clientes los experimenten, la calidad es algo que no se **debe** delegar.

El equipo de desarrollo deben seguir los siguientes principios :

- **Antes de programar, debe reflexionar sobre el uso** deseado del código por parte de los usuarios, los resultados esperados y los escenarios alternativos o no esperados a tener en cuenta.
- **Debe tomar la iniciativa de identificar y solucionar los errores** en su propio código antes de que se conviertan en problemas mayores. Esto implica realizar pruebas frecuentes y usar herramientas de monitoreo para detectar problemas de manera temprana.
- **Debe generar código que se pueda probar** automáticamente de manera fácil y efectiva. Por ejemplo, creando código poco acoplado, modularizable y cumpliendo entre otras cosas, los principios establecidos en el [playbook de desarrollo de código](#).
- **Deberían escribir pruebas** como parte del proceso de desarrollo para modelar el software basado en los requisitos, en el mismo sitio que el código de la aplicación, con sus mismas normas y su mismo ciclo de vida.
- **Deben** aplicar el principio “[boyscout](#)” y dejar el código (incluyendo los tests) mejor que lo encontraron.

El service owner y el product owner **deben** velar por los principios mencionados anteriormente.

Los equipos **deben** ser capaces de reconocer cuando hay problemas en el software que han producido y trabajar para corregirlos. Esto implica tomar la responsabilidad de los errores y aprender de ellos para evitar cometer los mismos errores en el futuro, para lo que es vital realizar un seguimiento periodico de la evolución de los defectos y planificar acciones correctivas para su mejora.

Los equipos que todavía no cuentan con el conocimiento necesario o no tienen la madurez suficiente **pueden** integrar temporalmente personas especializadas en pruebas para ayudarles a conocer las mejores prácticas, realizar las primeras automatizaciones o buscar soluciones a problemas comunes alrededor de las pruebas.

Los equipos **deberían** escribir pruebas como parte del proceso de desarrollo para modelar el software basado en los requisitos y **pueden** seguir un enfoque de [desarrollo guiado por pruebas](#)

[\(TDD\)](#), en el que los requisitos se traducen primero en pruebas y luego se implementa el código para que supere las pruebas, ya sea usando estilo [Londres o Chicago](#). Cuando se utiliza [TDD](#), no se escribe ningún código funcional sin una prueba de apoyo. [TDD](#) puede aumentar la calidad del código porque el equipo se centra en el comportamiento esperado del código y aumenta la cobertura de las pruebas. A largo plazo, escribir pruebas antes de codificar puede ahorrar tiempo, ya que los desarrolladores detectan los errores y los problemas en una fase más temprana del proceso de desarrollo, antes de que sean más complejos y se tarde más en solucionarlos.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jira** (o herramienta similar si no es posible su uso): Servicio global para la gestión de proyectos.
- **Bitbucket**: Sistema principal y central del Banco para el versionado del código. Se establecerá un único sistema global a modo de servicio único.
- **Equipo local**: El entorno local de cada desarrollador debe soportar la adopción de esta práctica. Debe incluir todas las herramientas y librerías necesarias para poder desempeñar su trabajo (e.g., Git, python, JUnit, sonar).

Indicadores

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Evolución de defectos por criticidad y fase	Número de defectos por criticidad (e.g. alta, media, baja) y fase (e.g. desarrollo, release, producción) respecto al periodo anterior (e.g. trimestre, mes)
Tiempo de resolución de incidencias o defectos por criticidad y fase	Tiempo transcurrido desde la fecha de detección hasta la fecha de resolución de cada defecto.
Disponibilidad de los servicios	Porcentaje de disponibilidad del servicio en entornos previos y productivos*.
Trabajo correctivo frente a evolutivo	Porcentaje de tareas de trabajo que han sido correctivas entre el número de tareas totales y categorizado por criticidad (e.g., alta, media baja)
Calidad técnica del código por disciplina	Número de violaciones de las herramientas de análisis estático (e.g, código duplicado excesivo, errores críticos y similares) por disciplina (e.g. seguridad, operaciones, desarrollo, etc)

Enlaces

- <https://www.thoughtworks.com/insights/blog/testing/test-driven-development-best-thing-has-happened-software-design>

2.3. Automatizar lo que se pueda para probar de forma eficiente, evidenciable, y detectar errores pronto

Beneficios

Los beneficios derivados del uso de una mezcla de tipos de pruebas que equilibren la velocidad de respuesta y la exhaustividad son:

- Detección más temprana de los defectos, lo que reduce el coste, el tiempo y el esfuerzo necesarios para solucionarlos.
- Mejora de la rentabilidad de las pruebas.
- Red de seguridad exhaustiva, que garantiza que todos los aspectos del sistema se prueban a fondo con diferentes tipos de pruebas, lo que reduce el riesgo de que los errores y los problemas pasen desapercibidos

Precondiciones

N/A

Adopción

2.3.1. Automatizar las pruebas que lo requieran

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita por grado de aplicabilidad, alineado con las directrices de la práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Build automatizada Trazabilidad tests-defectos Código-pruebas juntos Soporte pruebas Soporte IDE Entornos y pruebas locales Pruebas pre-deploy Enforcement merge Descarga datos periódicas Carga de datos periódicas Buscador de datos	Pruebas configurables Pruebas bajo demanda Pruebas planificadas Trazabilidad e2e Convenciones Repositorios funcionales Pruebas pre-deploy efímeros Pruebas pre-deploy virtualizadas Pruebas pre-deploy compartidos Pruebas post-deploy Rollback automático Enforcement despliegues Versionado test-app-entornos Descarga datos bajo demanda Reserva de datos	Versionado SUT Pruebas impacto InnerSource repos InnerSource pipelines Cambios entre pruebas Canaries/Feature Toggles Interceptación sistemas externos Trazas sistemas Monitorización y alertas

Adopción

El equipo **debe** conocer la [pirámide de pruebas](#) para seleccionar la combinación adecuada de pruebas en función de los requisitos del usuario final y de negocio, la adecuación al producto de software, la velocidad de respuesta, la exhaustividad, el mantenimiento y el coste de implementación.

Al seleccionar los tipos de pruebas y el nivel de automatización, los equipos:

- **Deben** identificar aquellas **pruebas repetitivas y automatizarlas**, dejando aquello que no sea repetitivo para los humanos (e.g., [pruebas exploratorias](#), [pruebas de aceptación](#)). Además, antes de subir a producción siempre **debe** existir alguna validación funcional del aplicativo por parte de un humano. En las pruebas exploratorias, se entrega a probadores cualificados un conjunto básico de instrucciones sobre el software y se les concede libertad para explorarlo, detectar defectos y notificar los resultados. Este enfoque permite a los equipos detectar defectos que no pueden descubrirse con pruebas automáticas.
- **Deben** procurar que la mayoría de los tests puedan ser **ejecutados en su entorno local de forma automática** de la forma más rápida posible, asegurando tener un buen balance entre valor aportado y tiempo extra requerido.
- **Deben** reutilizar las pruebas cuando fuera posible, p.ej. las pruebas e2e podrían reutilizarse para validar la misma funcionalidad en diferentes entornos o incluso como pruebas de integración utilizando la **virtualización**; evitando crear dos veces las mismas interacciones de interfaz de usuario.
- **Deberían** seguir el principio de "**shift left**", es decir, los desarrolladores deben invertir en pruebas que puedan ayudar a descubrir defectos en una fase más temprana del proceso de desarrollo con menos esfuerzo, en lugar de más tarde y con más esfuerzo, lo que reduce el tiempo y el coste totales necesarios para solucionarlos. Si una prueba de nivel superior (p.ej. una prueba e2e en Cells), reporta errores pero no hay pruebas de nivel inferior que fallen (p.ej. una prueba de integración o unitaria en ASO), los equipos **deberían** escribir una prueba de nivel inferior para poder detectar ese problema antes y con menor coste.
- **Deberían** emplear una **combinación de tipos de pruebas** específicas y otras más completas y avanzadas para verificar que los componentes individuales del software funcionan como se espera y que la aplicación cumple los requisitos empresariales y del usuario final en su conjunto.








		Análisis estático (cubierto en el playbook de "Análisis y Desarrollo de Código")			Tipos de pruebas dinámicas (cubiertas en este playbook)			
Pruebas de fiabilidad funcional		Lintor	Código estático y revisión de documentos		Pruebas unitarias	Pruebas de integración	Pruebas E2E/IU	Pruebas de aceptación
Pruebas no funcionales	Seguridad	AST estático	SCA	Política como código	AST Dinámico	AST Interactivo	Pruebas de penetración	
	Resiliencia						Pruebas de carga/estrés	Chaos Engineering
	Experiencia de usuario						Pruebas de compatibilidad	Pruebas de accesibilidad
Velocidad de Feedback			Rápido					Lento 
Amplitud			Centrado					Amplio 
Coste			Bajo					Alto 
 "Shift left" to increase testing efficiency								

Figura 2 - Comparación de distintos tipos de pruebas

Las **pruebas automatizadas** **deben** incluirse en el repositorio de código (en la mayoría de casos en el mismo repositorio que contiene el código a probar, aunque pueden darse excepciones dada la complejidad de algunos aplicativos multicapa y multi-repo para el caso de pruebas de monitorización o E2E) y seguir las **mismas directrices de "código de calidad"** que el código de la aplicación principal. Las directrices a aplicar para código de calidad se detallan en el [playbook de code development and versioning](#). Además, las pruebas automatizadas deberían atenerse a las siguientes prácticas:

- **Probar una sola característica al mismo tiempo:** Cada caso de prueba **debe** comprobar una característica o aspecto concreto de la aplicación. Esto contribuirá a que sus pruebas sean específicas y fáciles de mantener. p.ej en para probar que una calculadora funciona correctamente es deseable probar que cada operador (suma, resta, etc) funciona correctamente de forma aislada en vez de intentar probar escenarios dónde se prueben todos los operadores de forma simultánea.
- **Crear pruebas independientes, deterministas y reproducibles:** Las pruebas **deben** arrojar el mismo resultado con independencia del entorno, de otras pruebas o de la secuencia en la que se ejecuten. Los resultados de las pruebas solo pueden cambiar si cambia la dependencia, el código subyacente ó los datos. Este principio ayuda a reducir el riesgo de errores y facilita el aislamiento y la resolución de problemas.
- **Crear pruebas inmunes a cambios:** Para que las pruebas automatizadas funcionen incluso en caso de cambios, se **debe** evitar que un sólo cambio en el código deje obsoletas múltiples pruebas. Las pruebas **no deberían** basarse en datos de implementación, como elementos particulares, estructuras internas, etc. p.ej se

deben evitar pruebas que dependan de elementos que no sean públicos o que no tengan una interfaz clara y estable.

- **Reutilizar el código de prueba:** El código de prueba **no se debe** copiar y pegar. En lugar de repetir los mismos pasos en varias pruebas, **deberían** crearse módulos reutilizables, p. ej., saltos de pantallas de login. Intentando además facilitar dónde sea posible la reutilización de pruebas entre geografías.
- **Rápidas:** Para incrementar la probabilidad de que los desarrolladores las ejecuten con mayor frecuencia.
- **Simples:** Fáciles de entender y mantener para reducir la probabilidad de introducir errores nuevos.
- **Legibles** y con nombres autoexplicativos para reflejar el escenario que están sometiendo a prueba; los casos de prueba ayudan a la documentación porque son ejecutables y no se desincronizan con el código. La necesidad de comentarios puede ser un indicio de que hay que simplificar la prueba.
- **Uso de un identificador de caso de prueba único:** Las pruebas **deben** usar un identificador único para el ámbito del repositorio, a fin de poder hacer un seguimiento de los resultados de las pruebas. (e.g., utilizando un prefijo en los nombres de los casos de prueba o etiquetas adicionales como @Id).
- **Separadas de los datos de pruebas:** Las pruebas **deberían** evitar datos hard-coded. En su lugar, **deberían** usarse datos provenientes de una herramienta para la búsqueda de datos de prueba (p.ej Data Manager) para recuperar los datos de pruebas, p.ej es probable que el usuario y la contraseña a utilizar en cada entorno sea diferente.
- **Agrupadas por área funcional:** Las pruebas **deberían** agruparse conforme al área funcional de aplicación cubierta para facilitar la actualización de casos de prueba relacionados al modificar el área, p. ej., usando etiquetas y deberían ser el reflejo de los requisitos del sistema acordados con los diferentes stakeholders.

Indicadores

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Esfuerzo dedicado a pruebas	Porcentaje del esfuerzo de desarrollo dedicado a pruebas.

Correctivos con defectos detectados por pruebas automáticas	Porcentaje de correctivos con defectos detectados por las pruebas automáticas por criticidad (e.g. alta, media, baja) y fase (e.g. desarrollo, release, producción).
Grado de shift-left	Porcentaje de los defectos detectados por pruebas automáticas que han sido detectados por pruebas “rápidas” (e.g. valorando más las pruebas unitarias y menos las e2e), categorizado por criticidad (e.g. alta, media, baja) y disciplina (e.g. seguridad, operación, desarrollo, etc).
Falsos positivos en pruebas automáticas	Porcentaje de ejecuciones de pruebas automáticas obligatorias que han generado falsos positivos.

Enlaces

- <https://martinfowler.com/articles/practical-test-pyramid.html>
- <https://martinfowler.com/bliki/ExploratoryTesting.html>
- [https://es.wikipedia.org/wiki/Pruebas_de_aceptaci%C3%B3n_\(inform%C3%A1tica\)](https://es.wikipedia.org/wiki/Pruebas_de_aceptaci%C3%B3n_(inform%C3%A1tica))

2.3.2. Integrar pruebas en los pipelines de construcción y despliegue

Beneficios

Las ventajas de integrar pruebas en los *pipelines* de construcción y despliegue son:

- Aumento de la robustez del código que se fusiona con el máster o se despliega a producción, reduciendo así el riesgo de tener que corregir defectos más adelante.
- Eliminación de la necesidad de activar pruebas manualmente, dejando así más tiempo para el desarrollo de características.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Build automatizada Pruebas configurables Trazabilidad tests-defectos Código-pruebas juntos Soporte pruebas Soporte IDE Entornos y pruebas locales Pruebas pre-deploy Enforcement merge Descarga datos periódicas Carga de datos periódicas Buscador de datos	Pruebas bajo demanda Pruebas planificadas Trazabilidad e2e Convenciones Repositorios funcionales Pruebas pre-deploy efímeros Pruebas pre-deploy virtualizadas Pruebas pre-deploy compartidos Pruebas post-deploy Rollback automático Enforcement despliegues Versionado test-app-entornos Descarga datos bajo demanda Reserva de datos InnerSource repos InnerSource pipelines	Versionado SUT Pruebas impacto Cambios entre pruebas Canaries/Feature Toggles Interceptación sistemas externos Trazas sistemas Monitorización y alertas

Adopción

Tanto las pruebas automatizadas como las manuales **deben** integrarse en los *pipelines* de construcción y despliegue definidos en el servidor de automatización, p. ej., Jenkins.

Los equipos más maduros **deberían** poder contribuir al código de los *pipelines* y adaptar las pruebas a sus necesidades.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jira** (o herramienta similar si no es posible su uso): Servicio global para la gestión de proyectos.
- **Bitbucket**: Sistema principal y central del Banco para el versionado del código. Se establecerá un único sistema global a modo de servicio único.
- **Jenkins**: Servidor de automatización único dónde alojar los pipelines de construcción e integración. Los tests automáticos se orquestan como parte del pipeline.

Indicadores

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
-----------	-------------

Aplicaciones con pruebas automáticas en los Pipelines	Porcentaje de repositorios con pruebas automáticas obligatorias dentro de un servicio
---	---

2.3.3. Priorizar la ejecución de pruebas con el mayor impacto en cada fase

Beneficios

Las ventajas de priorizar la ejecución de pruebas con el mayor impacto en cada fase:

- Reducción del riesgo de defectos mediante el uso eficiente de los recursos de pruebas
- Detección temprana de defectos, lo que reduce el coste, el esfuerzo y el tiempo necesarios para corregirlos, gracias al uso de pruebas más impactantes y de smoke tests en una fase más temprana.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita.

Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Build automatizada Pruebas configurables Trazabilidad tests-defectos Código-pruebas juntos Soporte pruebas Soporte IDE Entornos y pruebas locales Pruebas pre-deploy Enforcement merge Descarga datos periódicas Carga de datos periódicas Buscador de datos	Pruebas bajo demanda Pruebas planificadas Trazabilidad e2e Convenciones Repositorios funcionales Pruebas pre-deploy efímeros Pruebas pre-deploy virtualizadas Pruebas pre-deploy compartidos Pruebas post-deploy Rollback automático Enforcement despliegues Versionado test-app-entornos Descarga datos bajo demanda Reserva de datos InnerSource repos InnerSource pipelines Versionado SUT	Pruebas impacto Cambios entre pruebas Canaries/Feature Toggles Interceptación sistemas externos Trazas sistemas Monitorización y alertas

Adopción

Los equipos **deben** priorizar la ejecución de las pruebas que tienen **más probabilidades de detectar defectos** más rápidamente en cada fase o que más nos reducen el riesgo, porque los recursos y el tiempo de cada fase son limitados y las series de pruebas deben ejecutarse rápidamente para animar a los desarrolladores a ejecutarlas con mayor frecuencia.

Las ejecuciones de las pruebas se suelen priorizar en función de su velocidad, dando **más prioridad a los tipos de pruebas más rápidas**, como las unitarias, y menos la ejecución de las pruebas más lentas, como las end-to-end. Además dentro de una misma tipología, los desarrolladores también **deberían** crear diferentes grupos para priorizar la ejecución de ciertas pruebas frente a otras y evitar así ejecutar todas las pruebas y tener que esperar bastante tiempo para darse cuenta que todas fallan por un problema en el proceso de login.

Para eficientar al máximo el tiempo de ejecución se deberían seguir las siguientes normas:

- Los equipos **deben** priorizar la ejecución de **smoke tests** para verificar las funcionalidades más críticas en la construcción y detectar problemas graves antes de realizar pruebas de mayor calado. Por ejemplo, primero validar si la home y el login funcionan correctamente antes de ejecutar un conjunto de pruebas mayor. Los smoke tests **deben** ser un subconjunto de la serie completa de pruebas automáticas cubriendo únicamente las rutas de funcionalidad más críticas y básicas en colaboración con el responsable de producto.
- Los equipos **deben** ejecutar una serie de pruebas de **regresión** antes de la **generación de releases** para asegurar que los cambios del software no introducen defectos nuevos en la funcionalidad existente. La serie de pruebas de regresión es un subconjunto de las pruebas existentes. Las pruebas de regresión se **deben** pasar utilizando entornos de servicio reales y **no se debe** usar virtualización de servicios.
- Los equipos **deben** tener un conjunto de pruebas al menos estáticas, unitarias e integradas (virtualizadas o mockeadas) obligatorias a superar antes de fusionar su código con la rama principal o generar versiones finales. Este conjunto debe cubrir al menos los smoke tests y las pruebas de seguridad SAST.
- Los equipos **pueden** ejecutar **análisis de impacto de pruebas (TIA)** como parte del *pipeline* del servidor de automatización para determinar las pruebas que es más probable que se vean afectadas por los cambios y priorizar su ejecución.

Los desarrolladores también **deberían** poder ejecutar tests o suite de tests específicos a demanda si lo consideran oportuno en función del tipo de cambio.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jira** (o herramienta similar si no es posible su uso): Servicio global para la gestión de proyectos.
- [Test-Manager](#): Herramienta para almacenar el resultado de los tests.
- [XRay](#): Herramienta para integrar métricas y reporting de los tests.

Herramientas

N/A

Indicadores

N/A

Enlaces

N/A

2.4. Seguir las mejores prácticas para cada tipo de prueba

Los equipos deberían escoger los tipos de pruebas que mejor se adapten a sus necesidades. Se ha definido una [matriz donde se recogen todos los tipos de pruebas y un ejemplo](#) anexa con los distintos tipos de pruebas definidas en esta sección y ejemplos para estandarizar la taxonomía.

Tipo de test (según funcionalidad y cobertura)	Que es (ejemplo)	Que no es (ejemplo incorrecto)
Análisis estático código	Control de excepciones en el código para llevar a cabo una transferencia de forma estática (sin ejecutar el aplicativo)	Ejecutar el aplicativo en un entorno previo y validar situaciones no esperadas (e.g., llevar a cabo una transacción desde una cuenta sin fondos)
Pruebas Unitarias	Se tiene una función que calcula el interés que genera una cuenta de ahorro dado un capital y una tasa de interés fija. La prueba unitaria de esta función llamaría a esa función con distintos escenarios para probar distintas casuísticas.	Se ejecuta la aplicación para validar que el tipo de interés mostrado en la interfaz gráfica es correcto dada una serie de datos sintéticos
Pruebas UI (unitarias)	Verificar el correcto funcionamiento de la función de inicio de sesión del sistema bancario en línea.	Definir un test que valide la operativa de login y una transacción contra el backend del aplicativo bancario
Pruebas Integradas virtualizadas (código del repositorio con dependencias virtualizadas)	Desarrollar un test con que valida el comportamiento de un módulo de pago que depende de un servicio externo de procesamiento de pagos (dependencia a virtualizar / mockear)	Definir un test que valide de forma unitaria la función que formatea las fechas de las transacciones y las muestra en la zona horaria adecuada
Pruebas Integradas sin virtualización (código del repositorio con dependencias reales)	Desarrollar un test con que valida el comportamiento de un módulo de pago que depende de un servicio externo de procesamiento de pagos. En este caso la llamada al servicio externo se hace sobre el servicio real levantado en el entorno previo correspondiente	Definir un test que valide de forma unitaria la función que formatea las fechas de las transacciones y las muestra en la zona horaria adecuada
Pruebas Contract tests (integradas sin virtualización)	Un servicio de banca proporciona una API para realizar transferencias bancarias. El servicio tiene un contrato que define los parámetros de entrada y salida, el formato de los datos y las restricciones del servicio. En este caso el ejemplo de prueba de contrato sería una prueba automatizada que verifique que el servicio cumpla con las especificaciones definidas en el contrato.	Validar de forma manual que nuestra aplicación funciona correctamente en entornos previos cuando llama servicios externos
Pruebas End to End	Validar el comportamiento completa de la aplicación en un escenario de uso real, por ejemplo desde el inicio de sesión hasta la confirmación de una transferencia.	Validar de forma integrada el proceso de transacciones de nuestro aplicativo, sin tener en cuenta el inicio de sesión o la confirmación final de la transferencia
...

Figura 3 - Tabla resumen con ejemplos de tipos de pruebas

Beneficios

N/A

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita por grado de aplicabilidad, alineado con las directrices de la práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Build automatizada Trazabilidad tests-defectos Código-pruebas juntos Soporte pruebas Soporte IDE Entornos y pruebas locales Pruebas pre-deploy Enforcement merge Carga de datos periódicas Buscador de datos	Pruebas configurables Pruebas bajo demanda Pruebas planificadas Trazabilidad e2e Convenciones Repositorios funcionales Versionado SUT Pruebas pre-deploy efímeros/asilados Pruebas pre-deploy virtualizadas Pruebas post-deploy / compartidos Saltar controles seguridad Rollback automático Enforcement despliegues Versionado test-app-entornos	Pruebas impacto InnerSource repos InnerSource pipelines Cambios entre pruebas Canaries/Feature Toggles Interceptación sistemas externos Trazas sistemas Monitorización y alertas

Adopción

2.4.1. Análisis estático

Beneficios

Las ventajas de usar pruebas estáticas son:

- **Más rápidas y económicas:** las pruebas estáticas pueden ayudar a detectar errores en una fase más temprana del ciclo de desarrollo, antes incluso de la compilación o la ejecución del código. Esto puede ahorrar gran cantidad de tiempo y recursos al prevenir la propagación de los errores por la base de código, lo que exigiría pruebas intensivas y correcciones más adelante.
- **Pruebas rentables:** las pruebas estáticas se pueden realizar de forma rápida y sencilla, y se pueden automatizar para ejecutarse con carácter periódico.
- **Garantía de cumplimiento:** las pruebas estáticas ayudan a garantizar que el código cumple las normas de la industria o las disposiciones reglamentarias. Esto puede tener una importancia especial en sectores como el sanitario o el financiero, en los que el cumplimiento es crítico.

- **Perfectas para código legacy:** el código legado puede resultar difícil de comprobar con pruebas unitarias, ya que es posible que no se haya escrito pensando en la comprobación. Las pruebas estáticas contribuyen a detectar problemas en el código legado y facilitan la creación de pruebas unitarias para cambios futuros.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

Los equipos **deberían** llevar a cabo [análisis de código estático](#) para detectar defectos en el código (e.g., uso de variables no inicializadas) y mejorar la calidad (e.g., reducir código duplicado), creando código con menos defectos desde el inicio. En el [playbook de análisis y construcción de código](#) se describen en detalle las prácticas de análisis de código estático y cómo aplicarlas.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Sonar:** Herramienta para evaluar código fuente. Es software libre y usa diversas herramientas de análisis estático de código fuente como Checkstyle, PMD o FindBugs para obtener métricas que pueden ayudar a mejorar la calidad del código de un programa.

Enlaces

- [análisis de código estático](#)

2.4.2. Pruebas unitarias

Beneficios

Los beneficios que conllevan este tipos de pruebas son:

- **Bucle de feedback corto** en comparación con otros tipos de pruebas.
- **Coste de implementación bajo** en comparación con otros tipos de pruebas.
- **Confiabilidad en el código**, con una red de seguridad para el desarrollo rápido de características y [refactorización](#) de código sin introducir errores.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

Normalmente las pruebas unitarias son automáticas y **deben** ser ejecutadas en los primeros pasos del proceso de CI/CD.

Las pruebas unitarias se diseñan para ser ejecutadas aisladas del resto del código, por tanto se pueden emplear dobles de pruebas como [mocks, dummies, stubs o fakes](#). Cuando la función a validar no es pública, tiene una interfaz o dependencias que suelen cambiar bastante o el código tiene muchas dependencias complicadas de mockear **debería** refactorizarse el código para facilitar las pruebas unitarias y reducir el coste de mantenimiento. En caso de no ser posible se **debería** considerar si es mejor usar un estilo [“classicist”](#) y usar ciertas dependencias reales; o usar mejor una prueba integrada en una función de más alto nivel que abarque también el comportamiento deseado. Debe tenerse en cuenta que en aplicaciones legacy, las pruebas unitarias pueden ser difíciles debido al gran acoplamiento y la falta de modularidad y por lo tanto debe valorar comenzar con pruebas de integración para verificar que los diversos componentes del sistema funcionen correctamente en conjunto.

Las pruebas unitarias tienen un alcance limitado y suelen probar una porción del código. El número de estas pruebas **deben** superar en gran medida a cualquier otro tipo de prueba.

El código trivial no debería someterse a pruebas unitarias. A la hora de implementar pruebas unitarias, los ingenieros de software **deben** centrarse en verificar una ruta crítica que es importante para garantizar el valor empresarial, en lugar de probar código trivial para aumentar la cobertura. Por ejemplo, los *getters* y los *setters* **no deberían** someterse a pruebas unitarias, a menos que ejecuten validación de datos o reglas empresariales.

Herramientas

- **Framework de pruebas unitarias:** La definición de pruebas unitarias depende del lenguaje de programación, por tanto la implantación de esta práctica requiere una gran variedad de herramientas o librerías (e.g., Mocha, JUnit, unittest, Scala Test, WCT).
- **Mockeo:** depende del lenguaje de programación, por tanto la implantación de esta práctica requiere una gran variedad de herramientas o librerías (e.g [mockito](#))
- **Virtualización:** [vBank](#).

Enlaces

<https://martinfowler.com/articles/mocksArentStubs.html>

2.4.3. Pruebas de integración

Beneficios

Las ventajas de usar pruebas de integración son:

- **Generación de confianza en el sistema:** las pruebas de integración proporcionan un nivel de confianza adicional en el sistema mediante la comprobación de todo el conjunto, en lugar de componentes individuales aislados. Esto ayuda a garantizar que el sistema funcionará según lo previsto en producción.
- **Documentación del comportamiento del sistema:** las pruebas de integración sirven como forma de documentación del comportamiento del sistema. Ayudan a las partes interesadas a entender cómo funciona el sistema y cómo interactúan las distintas partes del mismo.
- **Pruebas más completas** que las pruebas unitarias, lo que permite la verificación del comportamiento de las integraciones, p. ej., flujos de datos, temporalización, sincronización, etc.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

Las **pruebas de integración** **deben** utilizarse para comprobar que todo el código del repositorio funciona según lo previsto e interactúa correctamente con dependencias externas, como API, bases de datos, etc.

Cuando las dependencias externas o los subsistemas no estén disponibles o sean inestables, las pruebas de integración **deben** usar **virtualización** para reducir la inestabilidad, incrementar la fiabilidad de las pruebas y permitir la ejecución previa al despliegue.

Las pruebas de integración **deberían** cubrir:

- **Las especificaciones y los requisitos críticos** proporcionados por su código a usuarios terceros como una caja negra, para garantizar que la nueva funcionalidad no rompa la retrocompatibilidad.
- Las **interacciones críticas** entre los distintos componentes o módulos de su código. Esto incluye la definición de cómo interactúan los componentes entre sí y qué datos se intercambian.
- **Pruebas limitadas** para verificar el modo en que las dependencias de la aplicación interactúan entre sí al virtualizar las interfaces.

También se **debería** usar un subconjunto de pruebas de integración para supervisar el estado de salud de las funciones críticas en los entornos de trabajo (e.g., login, ver estado de cuenta). Por lo general, si en esta actividad de supervisión se detecta algún error, el desarrollador que lo corrigió **debería** intentar escribir un caso de prueba de nivel inferior (por ejemplo una prueba unitaria o una prueba integrada de un componente) capaz de detectar este problema más temprano en el futuro.

Si tus únicas pruebas de integración son “amplias”, deberías considerar explorar el estilo “específico”, ya que es probable que mejore significativamente la velocidad de tus pruebas, la facilidad de uso y la capacidad de recuperación. Dado que las pruebas de integración específicas tienen un alcance limitado, suelen ejecutarse muy rápidamente, por lo que pueden realizarse en las primeras etapas del SDLC, proporcionando una retroalimentación más rápida en caso de que fallen.

Herramientas

- **Framework de pruebas integradas:** La definición de pruebas de integración depende del lenguaje de programación, por tanto la implantación de esta práctica requiere una gran variedad de herramientas o librerías (e.g., Mocha, JUnit,wdio/[acis](#), Cucumber, [BackendTesting](#), [Data-manager](#))
- **Mockeo:** depende del lenguaje de programación, por tanto la implantación de esta práctica requiere una gran variedad de herramientas o librerías (e.g [mockito](#))
- **Virtualización:** [vBank](#).

Enlaces

- <https://martinfowler.com/bliki/IntegrationTest.html>

2.4.4. Contract Tests

Las [pruebas de contrato](#) (Contract Tests) son un tipo de prueba de software que se centra en verificar que la comunicación e interacción entre diferentes componentes, módulos o

servicios cumplan con un conjunto predefinido de reglas o contratos. Estas pruebas ayudan a garantizar que los componentes involucrados en la comunicación se adhieran a las expectativas mutuas y funcionen correctamente juntos. Un ejemplo en el sector bancario sería probar la interacción entre un servicio de banca en línea y un servicio de procesamiento de pagos, asegurándose de que las solicitudes de pago y las respuestas se ajusten al contrato establecido, como la estructura de datos y los códigos de estado.

Beneficios

- **Reducción del acoplamiento entre componentes:** Los Contract Tests ayudan a identificar y prevenir problemas de compatibilidad entre componentes al definir de manera clara las expectativas y responsabilidades de cada uno.
- **Mejora de la calidad del software:** Al garantizar que los componentes cumplan con sus contratos, se reduce la posibilidad de errores y problemas de calidad en el software.
- **Aceleración del proceso de desarrollo:** Los Contract Tests permiten que los desarrolladores trabajen en paralelo y de manera independiente, ya que se aseguran de que las interfaces y los contratos estén definidos y documentados correctamente.
- **Facilitación de la integración continua:** Los Contract Tests son una herramienta útil para implementar y mantener la integración continua, ya que permiten detectar y resolver problemas de compatibilidad y calidad de manera temprana y automatizada.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

La mayoría de los sistemas modernos están compuestos por un conjunto de servicios que se llaman unos a otros usando una interfaz. Estas interfaces pueden tener diferentes formas y tecnologías. Las más comunes son:

- REST y JSON a través de HTTPS
- RPC usando algo como gRPC
- Construyendo una arquitectura basada en eventos usando colas

Para cada interfaz hay dos partes involucradas: el proveedor y el consumidor. El proveedor sirve datos a los consumidores. El consumidor procesa datos obtenidos de un proveedor.

Las [pruebas de contrato](#) automatizadas aseguran que las implementaciones en el lado del consumidor y del proveedor de un servicio se adhieran al contrato definido. Sirven como un buen conjunto de pruebas de regresión y aseguran que las desviaciones del contrato se puedan detectar temprano.

Al usar “*consumer driven contracts*” (CDC), los consumidores de una interfaz escriben pruebas que verifican la interfaz teniendo en cuenta todos los datos que necesitan de esa interfaz. El equipo consumidor luego publica estas pruebas para que el equipo que publica el servicio pueda obtener y ejecutar estas pruebas fácilmente.

El equipo proveedor ahora puede desarrollar su API ejecutando las pruebas de CDC. Una vez que pasan todas las pruebas, saben que han implementado todo lo que necesita el equipo consumidor.

Enlaces

- <https://martinfowler.com/articles/consumerDrivenContracts.html>

2.4.5. Pruebas de UI

Las pruebas de IU validan que la interfaz de usuario de la aplicación funciona correctamente.

La entrada del usuario debe desencadenar las acciones correctas, los datos deben presentarse al usuario, el estado de la interfaz de usuario debe cambiar según lo esperado.

A veces se dice que las pruebas de interfaz de usuario y las pruebas de extremo a extremo son lo mismo. Probar su aplicación de un extremo a otro a menudo significa utilizar la interfaz de usuario. Lo contrario, sin embargo, no es cierto.

La prueba de su interfaz de usuario no tiene que hacerse de manera integral. Dependiendo de la tecnología que utilice, probar la interfaz de usuario puede ser tan simple como escribir algunas pruebas unitarias para el código javascript frontend con el backend desconectado.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- [wdio](#) / [acis](#)
- Galatea (saucelabs, [browserstack](#) y similares)

2.4.6. Pruebas end to end

Beneficios

- **Pruebas exhaustivas de todo el sistema**, desde canales hasta mainframe, para garantizar el comportamiento correcto antes de liberar el código a producción.
- **Proporcionan un nivel de confianza adicional**, en la funcionalidad y la fiabilidad del sistema, y ayudan a detectar errores que otros tipos de pruebas podrían pasar por alto.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

Las pruebas **no deberían** tener conocimientos de las partes internas de la aplicación (caja negra), sus dependencias deben ser totalmente operativas.

Las pruebas [end to end \(E2E\)](#) **deben** utilizarse para comprobar que el sistema completo funciona según lo previsto (desde las interfaces de usuario hasta los backends), antes de liberar el código a producción.

Normalmente la serie de pruebas E2E **deben** completarse antes de cada release para asegurarse de que los nuevos cambios funcionan conjuntamente de forma correcta.

Los equipos **deberían** aplicar las siguientes prácticas para mejorar la eficiencia de las pruebas UI:

- **Inmunes a cambios:** para que las pruebas automatizadas funcionen incluso cuando hay cambios en la interfaz de usuario, las pruebas **no deberían** basarse en datos de implementación de la interfaz de usuario, como etiquetas de ubicación o estructura HTML para ubicar elementos de la IU. Las pruebas **deberían** utilizar [página objetos](#) y *data-testid* para reducir la repercusión de los cambios sobre la interfaz de usuario y evitar especialmente que un cambio en la aplicación deje obsoletas varias pruebas.
- **Reducir el acoplamiento:** utilizando [deep-linking](#) (en el caso de frontales) para permitir el acceso rápido a la página o para ver qué hay que comprobar y evitar así dependencias innecesarias.
- **Promover la reusabilidad:** mediante el uso de pruebas que tengan cierta flexibilidad para funcionar en distintos entornos ya sean virtuales, efímeros o compartidos o con diferentes conjuntos de datos con vistas a garantizar que el sistema se comporta según lo previsto bajo distintas condiciones.

- **Deterministas:** Evitar temporizadores o pausas; la pruebas **deberían** prepararse para funcionar correctamente en distintos equipos informáticos y entornos.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- [wdio](#) / [acis](#)
- [Cucumber](#)
- Galatea (saucelabs, [browserstack](#) y similares)

Enlaces

- https://es.wikipedia.org/wiki/Pruebas_de_software
- <https://martinfowler.com/bliki/PageObject.html>
- https://es.wikipedia.org/wiki/Enlace_profundo

2.4.7. Pruebas de aceptación

Las [pruebas de aceptación](#) son un tipo de prueba de software que se realiza para asegurar que un sistema o aplicación cumpla con los requisitos y expectativas del usuario final o cliente antes de su lanzamiento o implementación. Estas pruebas ayudan a confirmar que el producto es adecuado para su uso y funciona según lo especificado. Un ejemplo en el sector bancario sería probar una nueva función en una aplicación de banca móvil, como el escaneo de cheques, en la que los usuarios finales o representantes del cliente validan que el sistema funciona correctamente y es fácil de usar.

Beneficios

Las ventajas de usar pruebas de aceptación son:

- Validación de la **satisfacción de los requisitos empresariales**.
- Garantía de que el sistema **cumple las normas y reglamentos** pertinentes.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

Las **pruebas de aceptación** **deben** utilizarse como paso de validación final antes de desplegar una característica para todos los usuarios en producción, asegurándose de que el producto final satisfaga los requisitos empresariales. Estas pruebas **deberían** ser automatizadas y los usuarios (de negocio o equivalentes) **deben** revisar sus resultados para

la autorización de los despliegues a producción. Algunas de ellas podrían ser manuales y en las automáticas su implementación podría materializarse en pruebas unitarias, de integración o end to end.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- [XRay](#) y el resto de herramientas para la automatización de pruebas.

Enlaces

- https://es.wikipedia.org/wiki/Pruebas_de_validaci%C3%B3n

2.4.8. Pruebas de seguridad

Beneficios

Las ventajas de usar pruebas de seguridad son:

- Reducción del riesgo de violaciones de la seguridad.
- Cumplimiento de los requisitos del director de seguridad de la información (CISO).

Adopción

A continuación se enumeran todos los test de seguridad que se pretenden incluir en el ciclo de vida del desarrollo del SW. Cuando se alcance un nivel de madurez en la ejecución de los test que se muestran a continuación, se pasará a la fase de calibración de tiempos de compromiso para la ejecución de cada uno de ellos:

- **SAST**: Tests automáticos de seguridad enfocados a encontrar vulnerabilidades en el código fuente. **Deben** orquestarse cada vez que el desarrollador haga push de su código a su rama de funcionalidad. En ese momento se ejecuta una llamada a los analizadores de código estático que tras su análisis y posterior triaje para evitar falsos positivos por parte de los especialistas de seguridad elaborarán un reporte donde se indicarán los hallazgos encontrados para que sean resueltos por parte del equipo de desarrollo. La nueva funcionalidad siempre debe ejecutar este tipo de pruebas y, si se encuentran vulnerabilidades críticas o altas tras la revisión hecha por parte de los equipos de seguridad, se bloqueará la integración del código con las ramas principales.
- **DAST**: Test automáticos de seguridad que se realizan a una aplicación cuando se está ejecutando. En el portal CISO debería ofrecer una lista completa de las pruebas DAST necesarias que se realizarán automáticamente antes de fusionar el código en la rama principal.

- **Hacking Ético:** pruebas de intrusión que idealmente deben realizarse de forma continua cada vez que se libere una funcionalidad de la aplicación. **Deben** realizarse antes de liberar cambios a producción para comprobar la seguridad de la aplicación mediante la simulación de un ataque funcional a la aplicación. Las pruebas de intrusión se llevarán a cabo por un equipo de seguridad interno u homologado por el CISO GSD. Si es posible, se usarán herramientas que automaticen parte o todo el test. Las pruebas de intrusión pueden durar entre unos minutos y varios días. El responsable de producto debe acordar con los especialistas de seguridad las ventanas idóneas para llevar a cabo el test siguiendo directrices generales del portal CISO. En el siguiente documento puede encontrar las [directrices detalladas de hacking ético](#).
- **SCA (Software Composition Analysis) / Supply Chain Security.** Cada vez más, los desarrolladores utilizan código Open Source o de terceros y esto genera un gran desafío para la seguridad del Banco. Por este motivo, se llevarán a cabo una serie de test automáticos que identifican dependencias de software open source en un código fuente dado. Este tipo de test sirve para evaluar la seguridad del uso de dicho componente open source o de terceros, la reputación, si el código está autorizado o restringido, la adecuación a licencias de uso y la calidad del código. También se comprueba la autorización de uso, la procedencia y la integridad.
- **Secretos:** Se ejecutarán una batería de test orientadas a detectar la existencia de secretos en los repositorios de código, las aplicaciones, los ficheros o cualquier otro elemento susceptible de ser mal usado reproduciendo esta mala práctica. Estos test son automáticos y estarán basados en expresiones regulares adaptadas para cada runtime, framework o lenguaje.
- **Imágenes:** Test orientados a detectar vulnerabilidades en el ciclo de vida de contenedores. Se llevarán a cabo escaneos de las imágenes y funciones serverless durante el proceso de construcción y despliegue en los contenedores.
- **Casos de Abuso.** A través del Modelado de Amenazas (Threat Modeling) que se propone en la práctica de código seguro del [Playbook de desarrollo y versionado de código](#), donde junto a los ingenieros de seguridad, los desarrolladores elaboran una serie de potenciales ataques al producto, se elaborará una serie de test que pretenden emular los casos de abuso definidos para garantizar que no son explotables los caminos definidos. En la medida de lo posible, se automatizarán estos casos de abuso para ejecutarlos antes de la subida a producción.
- **Integridad de elementos sensibles.** Adicionalmente, existirá un catálogo de componentes catalogados como críticos, los cuales tendrán una monitorización especial para evitar su manipulación y posterior uso sin revisión de seguridad. Para

garantizar la integridad de dichos componentes certificados, se llevarán a cabo test de integridad para verificar la inmutabilidad del código.

Idealmente todo test mencionado anteriormente será automático. No obstante, hay tareas que se deben llevar a cabo manualmente para garantizar la calidad y profundidad del análisis. **Cualquier hallazgo catalogado con severidad crítica o alta, bloqueará la integración con la rama principal hasta que sea corregido.** La función CISO GSD definirá los requisitos y los criterios por los cuales el código debe someterse a los diferentes test de seguridad y la frecuencia.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **SAST:** Chimera (que internamente usa las herramientas comerciales Kiuwan y CheckMarx). Para entorno local, a definir por CISO GSD.
- **DAST:** Herramienta(s) a definir por CISO GSD.
- **IAST:** Herramienta(s) a definir por CISO GSD.
- **RASP:** Herramienta(s) a definir por CISO GSD.
- **Pruebas de intrusión / hacking ético:** Las herramientas utilizadas en las pruebas de intrusión varían mucho. Generalmente, se utiliza una combinación de herramientas automáticas y manuales.
- **SCA (Software Composition Analysis):** Chimera (que internamente usa la herramienta open-source Dependency Track). Para entorno local, a definir por CISO GSD.
- **Detección de secretos:** Chimera (que internamente usa la herramienta in-house Secrets).
- **Imágenes (seguridad en contenedores):** Chimera (que internamente usa la herramienta comercial Prisma Cloud).

Enlaces

- [https://owasp.org/www-community/controls/Static Code Analysis](https://owasp.org/www-community/controls/Static_Code_Analysis)
- [https://en.wikipedia.org/wiki/Dynamic application security testing](https://en.wikipedia.org/wiki/Dynamic_application_security_testing)

2.4.9. Pruebas de rendimiento

Beneficios

Las ventajas de usar pruebas de rendimiento son:

- Verificación de la satisfacción de los requisitos de rendimiento.

Adopción

Las [pruebas de rendimiento](#) **deben** utilizarse para verificar que las aplicaciones de software pueden gestionar la carga de trabajo prevista dentro de unos parámetros de rendimiento aceptable y notificar indicadores de rendimiento claros.

Las pruebas de rendimiento **deberían** automatizarse. Las pruebas de rendimiento **deberían** realizarse en el *pipeline* antes de desplegar el código a producción y en iniciativas de pruebas de rendimiento especiales más extensas en paralelo al proceso habitual de desarrollo de características.

El equipo **debería** implementar las pruebas que se hayan acordado relevantes de este tipo por la organización o de forma conjunta con el responsable del producto en la práctica 2.1.

La lista completa de directrices de rendimiento para aplicaciones del Banco internas y orientadas al cliente está disponible en [norma OP-023](#).

Los equipos que gestionan infraestructura **pueden** utilizar **Chaos Engineering** para reforzar la resiliencia y la fiabilidad de la aplicación mediante la introducción intencionada de varios tipos de fallos, errores y anomalías, y la evaluación del modo en el que el sistema responde a ellos. El equipo **debería** definir la necesidad y la frecuencia de los ejercicios de ingeniería de caos.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Pruebas de carga:** Neoload, Loadrunner, Jmeter, DQ Portal, PMaaS.
- **Pruebas de resistencia:** Herramienta(s) a definir
- **Ingeniería de caos:** Herramienta(s) a definir

Enlaces

- https://es.wikipedia.org/wiki/Pruebas_de_rendimiento_del_software

2.4.10. Pruebas de accesibilidad

Beneficios

Las ventajas de usar pruebas de la experiencia de usuario son:

- Verificación de la usabilidad por parte de personas con discapacidad
- Verificación de la compatibilidad en distintos entornos (p. ej., navegadores, sistemas operativos, dispositivos)

- Alineamiento con la misión¹ y estrategia² del Banco.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

Las **pruebas de accesibilidad** **deberían** utilizarse para evaluar si las personas con discapacidad pueden usar la aplicación de software, p. ej., personas con discapacidad visual. Todas las aplicaciones del Banco **deberían** seguir la norma [WCAG2.1](#) y nivel AA. Las pruebas de accesibilidad **deberían** automatizarse y ejecutarse antes del pre-despliegue, adicionalmente este tipo de pruebas **pueden** complementarse de forma manual por perfiles con discapacidad para garantizar la accesibilidad.

Las **pruebas de compatibilidad** **pueden** utilizarse para **funciona** según lo previsto en distintos entornos, como navegadores, sistemas operativos y dispositivos. Los requisitos de compatibilidad **deberían** definirse junto con el responsable de producto y las pruebas **deberían** implementarse en consecuencia. Las pruebas de compatibilidad **deberían** automatizarse y ejecutarse antes de liberar código a producción.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- [Axe](#): Herramienta para facilitar las pruebas de accesibilidad.

Enlaces

- https://es.wikipedia.org/wiki/Accesibilidad_web
- <https://www.w3.org/WAI/standards-guidelines/wcag/glance/es>
- https://es.wikipedia.org/wiki/Pruebas_de_compatibilidad

2.4.11. Pruebas de monitorización sintética

Beneficios

¹ Misión el Banco: Poner al alcance de todos las oportunidades de esta nueva era.

² Inclusividad como parte de la estrategia de sostenibilidad.

- **Detección temprana de problemas:** La monitorización sintética permite identificar proactivamente problemas de rendimiento y disponibilidad antes de que afecten a los usuarios reales.
- **Medición del rendimiento en condiciones controladas:** Al simular interacciones de usuario y escenarios específicos, la monitorización sintética permite medir el rendimiento del sistema en un entorno controlado y estable, lo que facilita la comparación y el análisis de los datos.
- **Evaluación de acuerdos de nivel de servicio (SLA):** La monitorización sintética es útil para evaluar y garantizar el cumplimiento de los acuerdos de nivel de servicio establecidos con los proveedores y clientes.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la sub-práctica descrita por grado de aplicabilidad, alineado con las directrices de esta sub-práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada.

Adopción

Las service owner **debe** identificar la operativa crítica para sus clientes y desarrolladores y utilizar pruebas e2e o integradas para monitorizar su funcionamiento en todos los entornos posibles (productivos y previos) así como atender las alertas que se generan para realizar planes de acción y para restablecer el servicio lo antes posible cuando sea necesario.

Además, para cada alerta o defecto atendido por un equipo, este **debería** intentar crear una prueba automática que permita en el futuro detectar este tipo de problemas antes en el ciclo de vida de desarrollo, por ejemplo si el problema concreto lo generó un servicio de ASO se podría intentar implementar una pruebas unitaria o una prueba integrada que se pueda añadir a los pipelines de CI/CD.

Indicadores

N/A

Herramientas

N/A

Enlaces

- <https://martinfowler.com/testing/>

- [https://es.wikipedia.org/wiki/Pruebas de humo](https://es.wikipedia.org/wiki/Pruebas_de_humo)
- [https://es.wikipedia.org/wiki/Pruebas de regresión](https://es.wikipedia.org/wiki/Pruebas_de_regresi%C3%B3n)

2.5. Mantener los entornos compartidos estables

Beneficios

Las ventajas de mantener los entornos de pruebas compartidos estables son:

- Mejorar la productividad del resto de los equipos al evitar o reducir las pérdidas de servicio por cambios no probados.
- Velocidad a la hora de probar debido a que encuentras un entorno funcional.
- Mejores ciclos de feedback, ya que permiten encontrar los defectos en fases más tempranas.
- Mejor capacidad para reproducir errores.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita por grado de aplicabilidad, alineado con las directrices de la práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Build automatizada Pruebas planificadas Trazabilidad tests-defectos Pruebas post-deploy / compartidos Saltar controles seguridad Rollback automático Carga de datos periódicas Buscador de datos	Pruebas configurables Pruebas bajo demanda Trazabilidad e2e Convenciones Repositorios funcionales Código-pruebas juntos Soporte pruebas Soporte IDE Entornos y pruebas locales Pruebas pre-deploy Enforcement merge Versionado SUT Pruebas pre-deploy efímeros/asilados Pruebas pre-deploy virtualizadas Enforcement despliegues Versionado test-app-entornos Cambios entre pruebas Trazas sistemas Monitorización y alertas Carga de datos bajo demanda Reserva de datos Usuarios productivos	Pruebas impacto InnerSource repos InnerSource pipelines Canaries/Feature Toggles Interceptación sistemas externos

Adopción

2.5.1. Mantener los entornos compartidos estables

La motivación de esta práctica es permitir que los entornos previos (incluido producción) tengan un nivel de servicio y calidad similar al propio entorno de producción.

Los responsables de los servicios **deben** contar con un conjunto de **pruebas de monitorización sintética** (e2e e integradas) contra los entornos que permitan validar la calidad de su servicio al menos en entornos previos. Ante fallos, se **debe** priorizar su resolución y luego se **deberían** impulsar acciones o pruebas de más bajo nivel que eviten la repetición de problemas.

A nivel de entorno y dominio (e.g., factorías) se **debe** definir una persona de las principales áreas **infraestructura, desarrollo y arquitectura** que se encarguen de coordinar la resolución de los problemas en función de su tipología, llevando un historial de problemas y haciendo responsables a los diferentes equipos que más inestabilidad generen de los planes de mejora.

En caso en que un equipo despliegue en el entorno una versión que no funcione correctamente **deben** hacer **rollback** de la misma dejando en el entorno la versión funcional más actualizada, los procesos para que esto ocurra deberían ser automáticos, tal como se describe en el [playbook de despliegue y observabilidad](#)

Los equipos **deberían** mantener estables los datos de los entornos compartidos, en caso de una prueba los datos deberían devolverse a su estado original.

Indicadores

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Indisponibilidad de los entornos previos (no productivos)	Porcentaje del tiempo que el servicio ha estado caído en un entorno por nivel de impacto (e.g, total, alto, medio, bajo)

Herramientas

N/A

Enlaces

N/A

2.6. Medir resultados e impulsar acciones de mejora de forma continua

Beneficios

Las ventajas de hacer un seguimiento de errores, abordar pruebas erróneas y medir la eficiencia de los casos de prueba son:

- Identificación de áreas del software con oportunidades de automatización, lo que permite a los equipos de desarrollo centrar sus esfuerzos en el abordaje de estas áreas.
- Prevención de la acumulación o el agravamiento de defectos, lo que reduce el coste, el tiempo y el esfuerzo necesarios para corregirlos.
- Los indicadores de la eficiencia de los casos de prueba ayudan a identificar qué casos de prueba son más efectivos, lo que permite la asignación de recursos a las áreas más críticas de la aplicación.
- Aumento del sentido de responsabilidad con respecto a los defectos, lo que promueve el cumplimiento de mejores prácticas en una fase más temprana del proceso de desarrollo de software.

Precondiciones

Se han definido una serie de precondiciones necesarias para aplicar la práctica descrita por grado de aplicabilidad, alineado con las directrices de la práctica. Para más información puede consultar la [matriz de precondiciones del anexo](#) con todas las precondiciones y su definición detallada:

Debe	Debería	Puede
Build automatizada Pruebas planificadas Trazabilidad tests-defectos Código-pruebas juntos Soporte pruebas Pruebas locales Pruebas pre-deploy Enforcement merge Pruebas pre-deploy efímeros/asilados Pruebas pre-deploy virtualizadas Pruebas post-deploy / compartidos Saltar controles seguridad Rollback automático Enforcement despliegues Carga de datos periódicas Buscador de datos	Pruebas configurables Pruebas bajo demanda Trazabilidad e2e Convenciones Repositorios funcionales Soporte IDE Versionado SUT Versionado test-app-entornos Trazas sistemas Monitorización y alertas Carga de datos bajo demanda Reserva de datos Usuarios productivos de pruebas	Pruebas impacto InnerSource repos InnerSource pipelines Cambios entre pruebas Canaries/Feature Toggles Interceptación sistemas externos

Adopción

2.6.1. Medir resultados e impulsar acciones de mejora de forma continua

Los desarrolladores **deben** revisar la calidad de todas sus pruebas (unitarias, integradas, etc) y el resultado de las pruebas fallidas en cuadros de mando para hacer un seguimiento de la ejecución de distintas series de pruebas y su resultado (p. ej., en Test-Manager).

Con cada defecto no detectado por las pruebas el equipo **debería** crear nuevas pruebas automáticas para adelantar la detección de problemas similares en el futuro o adaptar las existentes.

Periódicamente los responsables del servicio y de los productos **deberían** revisar la evolución de sus indicadores y proponer planes de mejora en caso de ser necesario.

Para monitorear la efectividad de cada caso de prueba, los equipos de desarrollo **deben** implementar un **mecanismo de trazabilidad que vincule los tests y los defectos detectados por cada uno**. Se recomienda adoptar convenciones en los mensajes de commit para establecer una conexión clara entre las soluciones implementadas y las pruebas que identificaron los errores. Por ejemplo, el mensaje de commit "Fix: Corrige el cálculo de impuestos (TEST-123)" señala que se ha solucionado un error en el cálculo de impuestos y que el caso de prueba "TEST-123" fue el responsable de detectar dicho problema. Además, **debe** existir un inventario de defectos y tests fallidos accesible por los equipos de desarrollo.

Si una prueba fracasa en una rama de integración (*trunk, develop, release, master, etc.*), entonces:

- **Notificar el problema en la herramienta de gestión de backlog (e.g., Jira):** La persona que inicializa la ejecución del *pipeline* **debe** notificar el problema y vincular el caso de prueba asociado, el entorno y la criticidad. En las [definiciones del flujo de trabajo Jira](#) se ilustra el flujo de seguimiento de errores en Jira.
- **Poner la prueba «inestable» en cuarentena:** Si la prueba arroja resultados incoherentes (prueba «inestable») en varias ejecuciones, el equipo de forma consensuada con los responsables del producto o del servicio, **puede** poner en cuarentena la prueba con la aprobación del Tech Lead. El problema se **debe** abordar lo antes posible, para evitar la acumulación de grandes cantidades de pruebas puestas en cuarentena.
- **Solucionar la dependencia rota:** Para pruebas en fase de integración (p. ej., smoke tests) la virtualización de servicios **debería** utilizarse para mejorar la estabilidad de las dependencias y aquellos problemas de integración cuyo origen no esté deberían notificar al

Env Cops (visitar el [documento enlazado](#) para más información sobre el rol) para ayudar a su inventario y resolución.

Los equipos **deben** medir la eficiencia de los casos de prueba, promover los casos de prueba más eficientes en la suite de los smoke tests y planes de mejora de forma periódica.

Herramientas

La adopción de esta práctica requiere el uso de las siguientes herramientas:

- **Jira** (o herramienta similar si no es posible su uso): Servicio global para la gestión de proyectos.
- [Test-Manager](#): Herramienta para almacenar el resultado de los tests.
- [XRay](#): Herramienta para trazar los resultados de los tests.

Indicadores

Los indicadores que apliquen, **deben** ser proporcionados a los equipos para conocer el grado de madurez respecto a la práctica:

Indicador	Descripción
Operativas crítica con pruebas automatizadas	Porcentaje de operativa crítica con pruebas de monitorización sintética en entornos (previos y productivos*) y con los criterios de aceptación/validación acordados con negocio.
Fallos en la monitorización de la operativa crítica	Porcentaje de ejecuciones fallidas de las pruebas de monitorización sintética en entornos (previos y productivos*).
Errores descubierto por las pruebas automáticas	Porcentaje de errores detectados por pruebas automáticas en cada fase (e.g., local, desarrollo, release, producción).
Equivalente a horas manuales ahorradas	Número con las horas equivalentes de lo que se hubiera necesitado para ejecutar manualmente todas las ejecuciones automáticas realizadas.

Enlaces

- <https://martinfowler.com/bliki/TestCoverage.html>

3. Proceso

La realización de pruebas y el control de calidad **debe** formar parte integral de todas las fases del ciclo de vida del desarrollo de software. Todas las pruebas **deberían** planificarse y ejecutarse lo antes posible en el ciclo de vida del desarrollo de software. El [documento enlazado](#) ofrece una descripción detallada sobre los roles involucrados en este proceso.

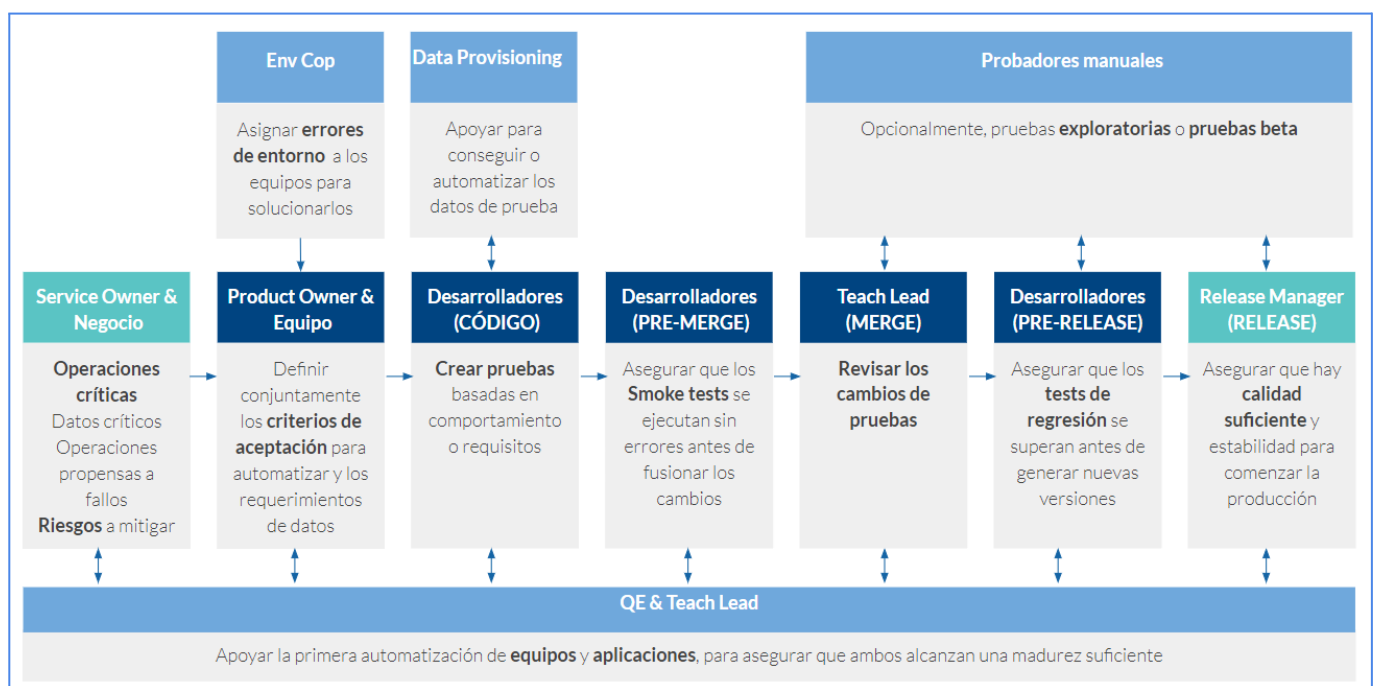


Figura 4: Ejemplo end to end del proceso de pruebas.

Proceso de realización de pruebas

1. Los QE Coach de forma conjunta con el Teach Lead **deberían** apoyar en las primeras automatizaciones de los equipos y de las aplicaciones para asegurar que ambos adquieran la madurez necesaria.
2. El Service Owner **debe** definir la operativa crítica, la operativa propensa fallos de su servicio o los riesgos más importantes a mitigar.
3. El Product Owner **debe** definir la funcionalidad, en particular los criterios de aceptación, los requisitos de datos de pruebas y las dependencias de servicios (consulte más información sobre este proceso en los playbooks de backlog management y de análisis y diseño). También debería definir qué funcionalidad de la que define es crítica.
4. El desarrollador **debe** diseñar e implementar pruebas durante el desarrollo de funcionalidad

- a. El desarrollador **debe** diseñar escenarios de prueba y casos de prueba basados en la historia de usuario.
 - b. El desarrollador **debe** decidir qué tipos de pruebas se utilizarán en la implementación de casos de prueba.
 - c. El desarrollador **debe** implementar pruebas automatizadas, además de diseñar y documentar instrucciones para casos de pruebas manuales.
5. El desarrollador **debería** probar código en su entorno local con pruebas unitarias y un subconjunto de pruebas de integración y E2E (end to end).
6. El desarrollador **debe** seguir las convenciones de commit definidas para tener trazabilidad de sus arreglos y los casos de prueba.
7. Se **deben** lanzar los smoke test antes de integrar código en la rama principal.
 - a. La apertura de una nueva solicitud de contribución **debe** activar automáticamente la ejecución de smoke tests. El objetivo de lanzar sólo los smoke tests es acotar los tiempos de ejecución y revisión de fallos, por tanto es esencial definir un buen conjunto de tests ejecutados en la suite de smoke tests.
 - b. El desarrollador **debe** corregir los errores descubiertos por los smoke tests antes de poder fusionar su código con la rama principal.
8. **Deben** realizarse pruebas de regresión antes y después de desplegar el código en producción.
 - a. La serie de pruebas de regresión se ejecuta automáticamente durante el proceso de liberación y **debe** incluir pruebas unitarias, de integración, E2E/IU y de aceptación. Además, la serie de pruebas de regresión **deberían** incluir un subconjunto de pruebas de seguridad, rendimiento y UX (cuando apliquen).
 - b. El desarrollador **debería** realizar pruebas manuales exploratorias. El desarrollador **puede** recibir la ayuda de un manual tester en este paso.
 - c. El usuario final o el Product Owner **debe** realizar pruebas de aceptación.
 - d. Todas las pruebas fallidas descubiertas en esta fase **deberían** notificarse en Jira.
 - i. El desarrollador **debe** corregir los errores introducidos por el nuevo cambio, e informar al Product Owner si los defectos no pueden corregirse durante el *sprint*.
 - ii. El desarrollador **debe** poner en cuarentena pruebas inestables con la aprobación del Tech Lead. Las pruebas inestables **deberían** eliminarse o corregirse lo antes posible.
 - iii. El equipo **debería** solucionar las dependencias rotas.

Anexo

Tipos de pruebas

En esta sección se va a presentar cuáles son los tipos de pruebas (categorías) que podemos emplear para validar que el código que desarrollamos se comporta correctamente. **No todas las pruebas aplican a todos los tipos de desarrollos, ni todos los equipos deben realizar todos los tipos de pruebas** (para más información visitar el capítulo [3.4. Seguir las mejores prácticas para cada tipo de prueba](#)).

El siguiente diagrama ilustra las diferentes fuentes y finalidades para las que surgen los diferentes tipos de pruebas. Principalmente las pruebas sirven a 4 propósitos:

- **Validar la funcionalidad del sistema:** la fuente de estos tipos de pruebas suele venir de los propios requisitos del negocio o del histórico de incidencias detectadas y/o reportadas, priorizados según su riesgo.
- **Eficientar la forma de asegurar la funcionalidad:** mediante el uso de pruebas automáticas que reemplacen aquellas pruebas manuales repetitivas.
- **Apoyar en el proceso de desarrollo:** el proceso de desarrollo se basa en el ensayo y el error (el desarrollador valida su avance hacia los criterios de aceptación conforme va codificando mediante pruebas lo más locales posibles a su cambio). Automatizar estas validaciones que se van realizando guiando el propio proceso de desarrollo y verificando el acercamiento a los criterios de aceptación permite no sólo hacer el proceso de desarrollo más estructurado y determinista, sino también contribuir a generar un set de activos de pruebas que contribuyan a asegurar la calidad y mantenibilidad del software generado.
- **Asegurar requisitos no funcionales:** orientados a garantizar los requisitos de rendimiento, de seguridad y legales entre otros y también priorizados según su riesgo.

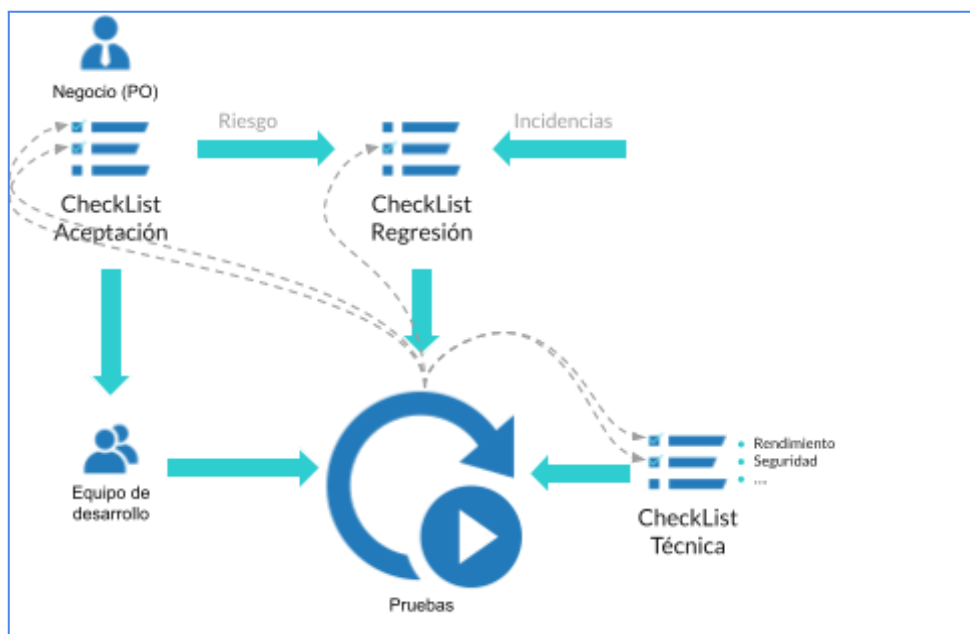


Figura 0.1 - Resumen ciclo de vida de pruebas y backlog

Es **crítico** para poder disponer de una correcta estrategia de pruebas, que los esfuerzos que se realizan durante el proceso de desarrollo de cara a un testing más proactivo, contribuyan a la [mitigación del riesgo](#) y a la **cobertura de los requerimientos** funcionales y no funcionales en la medida de lo posible. Dicho de otra manera, debería existir trazabilidad desde las pruebas que emergen desde los equipos de desarrollo hacia aquellos riesgos de negocio y/o requerimientos funcionales y no funcionales que contribuyan a cubrir. El equipo de desarrollo es responsable en primera persona de asegurar que dicha trazabilidad es correcta y de velar porque los tests continúan operativos durante la vida del software desarrollado.

A continuación, se define un resumen de los tipos de pruebas más populares que se tratarán a lo largo del playbook y sus respectivas clasificaciones. Es importante tener en cuenta que las diferentes clasificaciones no son exclusivas y una misma prueba puede pertenecer a varios tipos, por ejemplo podemos tener una prueba de aceptación tanto manual como automática.

- **Pruebas según su grado de cobertura:** Clasificación de las pruebas en función del "tamaño" o grado de integración del sujeto bajo prueba. Esta categorización es popular debido a su relación con costos y tiempos, y se alinea con la [pirámide de tests](#).
 - **Línea de código:** Pruebas enfocadas a analizar características propias del código de forma estática (i.e. evaluando cada línea de código).
 - **Unitarias:** Pruebas que se centran en una unidad individual de código, como una función o un método, p. ej probar una función que realiza la suma de dos números.
 - **Integradas:** Pruebas que evalúan la interacción entre componentes individuales dentro de un sistema, p. ej probar que el componente encargado de calcular las comisiones bancarias para diferentes tipos de transacciones, como transferencias

internacionales o retiros de cajeros automáticos, funciona correctamente y aplica las tarifas adecuadas. Ejemplos de este tipo de pruebas son las pruebas de contratos, las de componentes, o las integradas virtualizadas.

- **E2E:** Pruebas que evalúan el flujo completo de un sistema desde el inicio hasta el final, incluyendo todas las interacciones del usuario, capas y componentes, p. ej probar el proceso de compra en una tienda en línea desde la selección del producto hasta el pago y confirmación.

■ **Pruebas según el tipo de validación:** Clasificación de las pruebas en función del tipo de validaciones que haremos durante la prueba:

- **Funcionales:** Validar características y funcionalidades, p. ej probar que un sistema de autenticación permite el acceso a usuarios registrados.
- **No funcionales:** Evaluar aspectos como rendimiento, seguridad, usabilidad.
 - UX: Pruebas que evalúan la experiencia del usuario en términos de facilidad de uso, diseño y accesibilidad, p. ej probar la navegación y la disposición de los elementos en una aplicación web.
 - Seguridad: Pruebas que evalúan la robustez y seguridad del sistema frente a posibles amenazas y vulnerabilidades, p. ej probar la protección contra inyección SQL en una aplicación web.
 - Rendimiento: Pruebas que evalúan el comportamiento del sistema bajo carga y estrés, p. ej probar cómo responde un servidor web ante un gran número de solicitudes simultáneas.

■ **Pruebas según su enfoque:** Clasificación de las pruebas según el enfoque de la prueba:

- **Pruebas estructuradas (scripted):** Pruebas que siguen casos de prueba predefinidos. Ejemplo: probar el flujo de un proceso de registro siguiendo un conjunto de pasos y criterios preestablecidos.
- **Pruebas no estructuradas (no-scripted):** Pruebas exploratorias y basadas en la experiencia del probador.
 - Exploratorias: Pruebas que no siguen un plan predefinido y se basan en la experiencia e intuición del probador, p. ej investigar una aplicación para identificar posibles defectos y áreas problemáticas de manera libre y flexible.
 - Alpha/Beta: Pruebas realizadas por usuarios finales en entornos reales antes del lanzamiento oficial, p. ej invitar a un grupo de usuarios seleccionados a probar una versión beta de una aplicación.

■ **Pruebas según su momento de ejecución:** Clasificación de las pruebas según el momento de ejecución de las pruebas en el ciclo de vida del desarrollo:

- **Pruebas estáticas:** Revisión de código, inspecciones, [análisis estático](#), p. ej revisar el código fuente en busca de posibles errores o violaciones de las buenas prácticas de programación.
- **Pruebas dinámicas:** Son aquellas que se realizan mientras el código está en ejecución, tienen como objetivo asegurar que el desarrollo se comporte de acuerdo con los requerimientos del negocio (pruebas funcionales y no funcionales.)

- **Pruebas según su finalidad:** Clasificación de las pruebas según el objetivo que persiguen (e.g., seguridad, aceptación, regresión, monitorización, rendimiento).
- **Pruebas según su grado de automatización:**
 - **Manuales:** son pruebas que pudiendo estar apoyadas por herramientas o scripts, necesitan de intervención humana para su inicio, ejecución y/o evaluación de los resultados.
 - **Automáticas:** son aquellas que funcionan de forma desatendida siendo capaces de ejecutarse dentro del propio ciclo de vida del software, evaluar los resultados e intervenir en caso de encontrar problemas.

Los tipos de tests definidos en el playbook se detallan a continuación identificando algunas de las propiedades mencionadas previamente:

Tipo de test (según funcionalidad y cobertura)	Grado de cobertura	Funcionales	Estructuradas	Dinámicas	Automatizable
Análisis estático código	Línea de código		✓		✓
Pruebas Unitarias	Unitarias	✓	✓	✓	✓
Pruebas UI (unitarias)	Unitarias	✓	✓	✓	✓
Pruebas Integradas virtualizadas (código del repositorio con dependencias virtualizadas)	Integradas	✓	✓	✓	✓
Pruebas Integradas sin virtualización (código del repositorio con dependencias reales)	Integradas	✓	✓	✓	✓
Pruebas Contract tests (integradas sin virtualización)	Integradas	✓	✓	✓	✓
Pruebas End to End	End to end	✓	✓	✓	✓
Pruebas de Monitorización Sintética (e2e e integradas)	End to End / Integradas		✓	✓	✓
Pruebas de Aceptación (manuales y e2e)	Unitarias, integradas o End to End	✓	✓	✓	✓
Pruebas Exploratorias (manuales y e2e)	End to end	✓		✓	
Pruebas Alpha / Beta (manuales y e2e)	End to end			✓	
Pruebas de Rendimiento	Integradas o end to end		✓	✓	✓
Pruebas de Accesibilidad (con dependencias virtualizadas)	Unitaria, integrada y end to end	✓	✓	✓	
Pruebas de Seguridad SAST	Línea de código		✓		✓
Pruebas de Seguridad DAST	Unitaria, integrada y end to end		✓	✓	✓
Pruebas de Seguridad Hacking Ético	Unitaria, integrada y end to end			✓	

Figura 0.2 - Tipo de pruebas y propiedades

Habilitadores

Se han identificado una serie de habilitadores detallados a continuación que son necesarios para hacer viable la implantación de las prácticas propuestas en este playbook.

Entornos y dispositivos

Una de las condiciones necesarias para poder verificar el software es **contar con los entornos adecuados**. Los entornos de pruebas deberían ser, dentro de lo posible, [réplicas del entorno de producción](#), esto no se refiere al tamaño del entorno, sino a que las versiones, dependencias y herramientas que tengamos en los entornos de pruebas sean las mismas, siempre que se pueda, a las de producción. Además, aquellos equipos que por su naturaleza tengan que tratar con dispositivos físicos (e.g., teléfonos móviles, cajeros automáticos, tpvs) **deben** contar con acceso a los mismos (si la tecnología lo permite y el caso de uso lo respalda) durante la fase de pruebas para poder verificar que el software funcione correctamente en ellos.

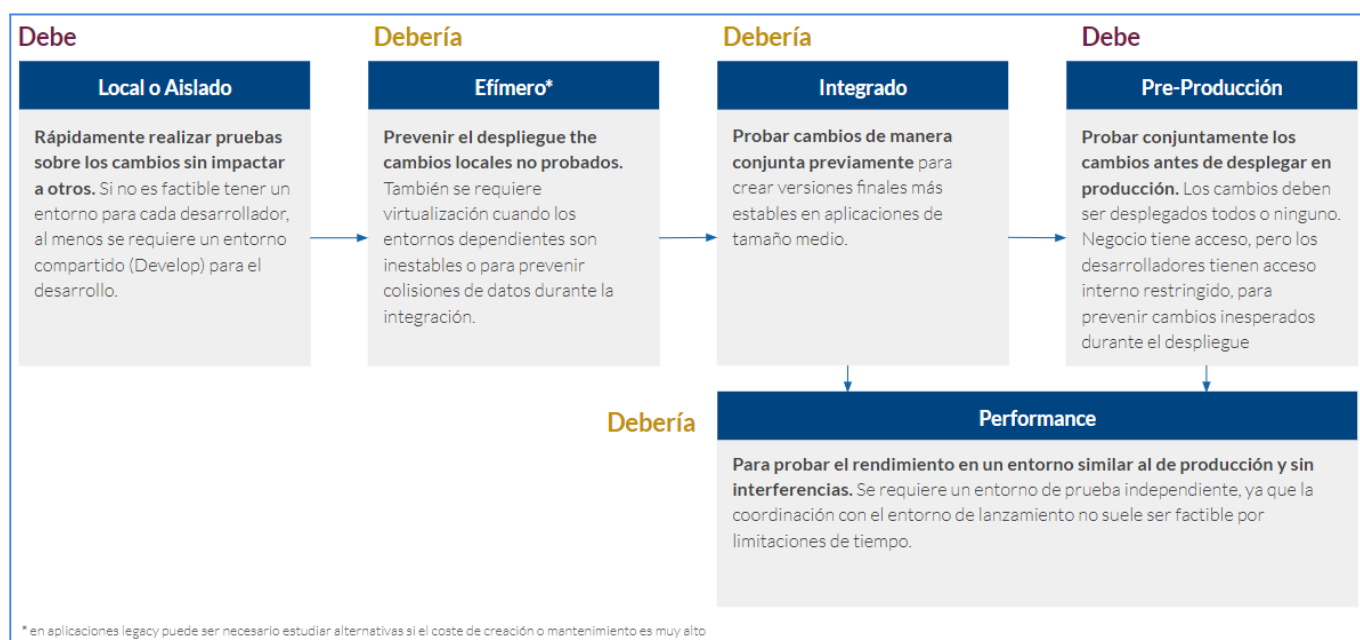


Figura 0.3 - Entornos previos

Los equipos que usan **estrategias de despliegue progresivas** pueden ser una excepción y pueden probar código directamente en producción utilizando pruebas **Alpha/Beta**, versiones **Canary** y procedimientos estrictos de supervisión siempre que el juego de pruebas automáticas les de la cobertura suficiente. Consulte el [playbook de despliegue y observabilidad](#) para obtener una descripción completa de las estrategias de despliegue.

Contar con entornos y dispositivos adecuados para pruebas nos permite:

- **Detección temprana de los problemas:** la realización de pruebas con datos y configuraciones similares a los de producción proporciona una representación más exacta de la forma en que el sistema se comportará en la vida real. Esto ayuda a detectar problemas lo antes posible haciendo que los ciclos de corrección de errores sean mucho más rápidos.

- **Capacidad de reproducir problemas fuera del entorno de producción:** Al tener unas condiciones similares a producción podemos reproducir incidencias más fácilmente, haciendo los tiempos de diagnóstico mucho más cortos.
- **Mejora del rendimiento:** la realización de pruebas con volúmenes, dispersión, datos y configuraciones similares a los de producción ayuda a detectar posibles cuellos de botella, lo que permite a los desarrolladores optimizar el sistema y garantizar el cumplimiento de los requisitos de rendimiento.

Idealmente podemos nombrar tres grandes grupos de entornos de prueba: **local o aislado, efímero y compartido persistente**. Los entornos de cada grupo serán detallados a continuación.

Para más información sobre la **frecuencia mínima** de ejecución de pruebas en los diferentes entornos se facilita la [matriz tests-entornos definida como anexo al playbook](#).

	Local / Aislado	Pipeline / Efímero*	Integrado	Preproducción	Rendimiento	Producción
Análisis estático código	Build	Push	-	-	-	-
Pruebas Unitarias	Build	Push	-	-	-	-
Pruebas UI (unitarias)	Build	Push	-	-	-	-
Pruebas Integradas virtualizadas (código del repositorio con dependencias virtualizadas)	Before Push*	Pull Request	-	-	-	-
Pruebas Integradas sin virtualización (código del repositorio con dependencias reales)	OnDemand*	OnDemand	Post Deploy / Nightly	Post Deploy / Release / Nightly	-	-
Pruebas Contract tests (integradas sin virtualización)	OnDemand*	OnDemand	Post Deploy / Nightly	Post Deploy / Release / Nightly	-	-
Pruebas End to End	OnDemand*	OnDemand	Post Deploy / Nightly	Post Deploy / Release / Nightly	-	-
...

Figura 0.4 - Comparación de distintos tipos de pruebas

Entornos locales o aislados

El entorno de desarrollo local es aquel entorno que se ejecuta en la máquina del desarrollador. En estos entornos los ingenieros de software realizan las **pruebas iniciales antes de integrar el código** en el control de versiones y desplegarlo a entornos remotos.

Los entornos locales permiten **trabajar de forma aislada**, lo que evita colisiones y **acelera la validación de los cambios** que se han realizado, sin interferencia con los cambios de otros. Sin embargo, los entornos locales están limitados a los recursos de hardware del equipo del desarrollador, el acceso a los datos y la integración de extremo a extremo. Estas limitaciones hacen que en ciertos casos únicamente se puedan ejecutar un subconjunto de pruebas.

Los análisis de código estático, las pruebas unitarias y una parte de las pruebas de integración deben poder ejecutarse en un entorno local. Se debe usar un subconjunto del conjunto de datos de producción o datos de pruebas sintéticos que imitan la producción cuando la tecnología lo permita y que abarque los escenarios que se están probando.

Para más información puede consultar el apartado [2.1. Construir y validar de forma automática el software en un entorno local del Playbook de Code Analysis and Building](#)

En tecnologías en las que no es posible tener entornos locales, como en *mainframe*, los ingenieros de software deberían disponer de un mecanismo para codificar y probar sus cambios sin afectar a otros, por ejemplo entornos aislados.

Entornos efímeros

Los entornos efímeros son **entornos temporales** de prueba creados automáticamente y desechados tras su uso, permitiendo una mayor eficiencia y agilidad en el desarrollo y pruebas de aplicaciones.

Una característica adicional de este tipo de entornos es la posibilidad de generarlos como réplicas de otros entornos existentes (p.e. entorno efímero espejo de producción) lo que confiere mayores garantías sobre la reproducibilidad de las pruebas³. Si además es posible poblar el entorno efímero con datos equiparables o consistentes con los de producción, aumentará la fiabilidad o consistencia⁴ de los resultados obtenidos.

Un ejemplo de entorno efímero podría ser la creación de una instancia temporal de la plataforma de banca en línea para **validar cambios de forma aislada antes de implantarlos** o integrarlos en entornos compartidos o ramas que den servicio a muchos Developers y dónde un fallo impacte a muchas personas.

Para poder habilitar su uso a los equipos de Desarrollo, aquellos sitios dónde sea posible, los responsables de mantener la infraestructura deberían **escribir** la [infraestructura como código](#), p. ej., containers, Terraform, Ansible, etc., de una forma coordinada también con otros equipos como los responsables de las plataformas y similares y siguiendo las prácticas de versionamiento adecuadas como en el resto del código. Para más información se puede revisar el [playbook de despliegue y observabilidad](#)

En estos entornos deben tener un **balance entre rapidez y similitud con producción** y sus dependencias pueden ser tanto dependencias reales como fakes, dependiendo de las necesidades de cada caso.

La creación de un entorno efímero se realizará usando los *scripts* versionados como código. El entorno **debe** aislarse y ejecutarse en paralelo con otros entornos de prueba sin afectarlos. Las dependencias de servicios de un entorno efímero deberían definirse en código (incluida la versión).

Los entornos efímeros pueden ser usados por un desarrollador o grupo de desarrolladores para validar sus cambios. También el pipeline puede tener pasos automáticos que pueden crear estos

³ Los test diseñados y ejecutados en entornos previos (efímeros) se podrían ejecutar en Producción sin necesidad de modificar el código.

⁴ La ejecución de un mismo proceso con los mismos datos o conjuntos de datos consistentes genera salidas similares y/o consistentes

entornos efímeros para validar alguna de sus características y estos entornos deberían **hacer uso de la virtualización** cuando los entornos dependientes son **inestables** ó se quieran evitar **interferencias** con terceros, **consumos excesivos** ó colisiones de datos durante la integración.

Las pruebas que chequean requisitos no funcionales o destructivas (performance, reliability, etc..) se ven muy beneficiadas por este tipo de entornos, que al ser aislados permiten llevar al sistema hasta sus límites sin afectar al resto de las personas que están probando.

El equipo de desarrollo debe destruir los entornos efímeros que no esté usando.

Entornos compartidos persistentes

Los entornos compartidos persistentes son aquellos que se encuentran **activos todo el tiempo**. Al igual que los entornos anteriores su configuración debe ser lo más cercana a producción posible.

Estos entornos son utilizados por uno o varios equipos para desplegar las últimas versiones de los componentes o servicios y realizar las pruebas de sus componentes **con las últimas versiones de los componentes de otros equipos** que se van a liberar. Los entornos deben cumplir las siguientes necesidades:

- Debe existir un modelo estándar para el gobierno de esos entornos.
- Deben existir mecanismos para coordinar el uso de dichos entornos y evitar que los equipos entorpezcan el trabajo de otros.
- Deben permitir a los desarrolladores desplegar sus soluciones de software sin impedimentos y agilizar la validación del software.
- Deben ser estables, tanto en la versiones de las dependencias contra las se prueba como con los juegos de datos.

Normalmente, teniendo en cuenta su función nos encontramos con los siguientes entornos compartidos: **desarrollo, integrado, rendimiento, preproducción y producción**.

Desarrollo

Cuando no es posible tener un entorno local o efímero para cada desarrollador, los desarrolladores deben, como mínimo, disponer de un entorno compartido para desarrollar y comprobar que sus cambios funcionan según lo previsto antes de incluirlos en la liberación.

Integrado

También llamado entorno Test. Las aplicaciones de tamaño medio, para conseguir que el **tiempo de estabilización de la release** en el entorno de preproducción **sea aceptable**, deberían contar con un

entorno de integrado dónde los cambios se integran y el código se prueba con todos los componentes dependientes disponibles, pero sin efectos para los usuarios finales. Se usan datos de pruebas no sensibles.

Es necesario cuando se producen muchos problemas de integración durante la fase de aceptación.

Rendimiento

Entorno que **debe** tener la configuración de hardware más similar a la de producción y para **probar el rendimiento en él sin interferencias**, es necesario un entorno de prueba separado, ya que coordinar con el entorno de lanzamiento a menudo no es viable debido a limitaciones de tiempo.

Preproducción

También conocido como entorno de Release, de Aceptación de Usuario o de Calidad. Es un entorno que es necesario para comprobar que todos los cambios para **las releases funcionan correctamente de forma conjunta** y con el resto de las capas antes de su despliegue a producción por todos los stakeholders. El código se prueba en un entorno de pre-producción que se aproxima lo máximo posible al entorno de producción, salvo que no contiene datos sensibles. Este entorno se expone a las partes interesadas con fines de comprobación y contiene la siguiente release para desplegar al entorno de producción. Después del despliegue a producción, la fase de aceptación debe tener el mismo código que en producción. Los desarrolladores no deberían tener acceso interno o directo a este entorno para poder hacer cambios libremente en los datos y software para así evitar colisiones durante la ejecución de pruebas.

Estos entornos deben ser accesibles por los usuarios de negocio para hacer sus validaciones.

Producción

El entorno productivo es utilizado por los usuarios finales, mientras que los entornos previos son utilizados exclusivamente por el equipo de desarrollo y prueba para asegurarse de que el software se encuentre en su mejor estado posible antes de ser desplegado en el entorno productivo.

Este entorno, además de por el usuario final, puede ser accesible por perfiles concretos y cualificados dentro del equipo. En el entorno de producción se deberán ejecutar mecanismos de validación y despliegue muy concretos (e.g., pruebas de monitorización, canary release, A/B testing).

Datos de prueba

Un factor clave de la realización de pruebas es la disponibilidad de datos de pruebas. Los datos de pruebas deberían respaldar casos de prueba, ser **reusables entre ejecuciones** y aproximarse lo

máximo posible a los datos de producción (identificando y anonimizando o enmascarando aquellos datos que sean sensibles) para incrementar las probabilidades de descubrir defectos antes de liberar código a producción. El equipo debería racionalizar la cantidad de datos necesarios y definir su conjunto de datos de prueba mínimo para dar cobertura a sus casos de prueba.

Los equipos deberían contar con mecanismos para realizar reservas de datos para las pruebas y se les deben facilitar mecanismos para tener un gobierno global adecuado de los mismos que garantice la accesibilidad.

El equipo debe determinar los requisitos de los datos de pruebas en la fase de diseño de las pruebas para no retrasar la implementación más adelante. Los equipos deberían usar los siguientes enfoques para generar datos de pruebas:

- **Creación de subconjuntos de datos y enmascaramiento** para crear buenos datos de pruebas a partir de los datos de producción, ya que este enfoque es rentable y **deben** eliminar datos sensibles para cumplir la normativa (p. ej., con TDM como Icaria, Ab Initio, Broadcom Test Manager o similares).
- **Golden copies (únicas) y virtualización** para crear un subconjunto de datos estable, determinista y reutilizable con vistas a garantizar la coherencia y la exactitud de los datos que también puede ser utilizado eficazmente por entornos efímeros (p. ej., Delphix, Accelario y similares).
- **Datos sintéticos** para crear datos a demanda para escenarios y condiciones específicas, lo que evita colisiones, permite realizar pruebas coherentes y reproducibles y nos evita posibles problemas de confidencialidad (p. ej., usando transacciones de *mainframe*, API de servicios, generadores de datos y otras herramientas para crear datos sintéticos).

Los equipos deberían crear una **taxonomía o base de conocimiento** con los datos necesarios para sus pruebas, su priorización y cómo es su proceso de creación, maduración y mantenimiento. Por ejemplo, usuarios con tarjetas, usuarios con préstamos, con depósitos, etc., siendo recomendable que esa taxonomía sea lo más global posible para que puedan servir para las aplicaciones globales.

Los equipos también deberían promover la **creación y la publicación de esos datos** en una herramienta accesible para pruebas manuales y automatizadas (p. ej., Data-manager en el Banco) en la que todos puedan encontrar y reservar, si es necesario, usuarios de cada tipo en la taxonomía para un entorno específico.

Los siguientes enfoques deberían utilizarse para reducir la corrupción de los datos:

- **Actualizar o limpiar los datos** con la mayor frecuencia posible para las pruebas automáticas; lo ideal es limpiar los datos antes de cada ejecución de pruebas, ya sean manuales o automáticas.
- **Aislar consumidores:** evitando la reutilización de los mismos datos en distintas unidades, p. ej., mediante la creación de distintos espacios de nombres en Data-manager para cada uno de ellos.
- **Coordinar el uso de los usuarios:** evitando el uso de los mismos datos para distintas pruebas al mismo tiempo o que su información en Data-Manager se quede obsoleta.
- **Prevenir cambios indeseados:** restringiendo el acceso de los desarrolladores a los entornos para evitar modificaciones de datos y despliegues de cambios nuevos durante la ejecución de pruebas.
- **Controlar los despliegues:** promoviendo despliegues pequeños y utilizando distintos días para distintas pilas tecnológicas y otras técnicas con vistas a minimizar conflictos y problemas de coordinación.

Cuando sea posible las aplicaciones deberían incluir los scripts, datos o cualquier otra información necesaria para su aprovisionamiento de los datos de prueba en el repositorio de código fuente. Sin embargo, si los datos de pruebas tienen un tamaño sustancial o se comparten entre varias aplicaciones, entonces debería evaluarse su gestión por un equipo centralizado de gestión de datos en un repositorio o aplicación de datos de pruebas especial.

Las ventajas del uso de datos de pruebas relevantes y de máxima calidad son:

- **Mejora de la eficacia de las pruebas:** los datos de pruebas de máxima calidad ayudan a los probadores a detectar defectos y errores en el software de una forma más eficaz.
- **Reducción de los costes de las pruebas:** mediante una gestión eficaz de los datos de pruebas, el coste de las pruebas puede reducirse al minimizar el tiempo y el esfuerzo necesarios.
- **Apoyo de la automatización de las pruebas:** la gestión de los datos de pruebas es crítica para la automatización de las pruebas, ya que garantiza que las pruebas automatizadas se ejecuten con los datos de pruebas correctos.

Mejora de la calidad general del software: unos datos de pruebas pertinentes y exactos permiten mejorar la calidad general del software.

Matriz precondiciones

Además de los habilitadores mencionados en esta misma sección. Se detallan a continuación las precondiciones (mecanismos que deben implantarse para poder adoptar las prácticas en su totalidad) identificadas para cada una de las prácticas de este playbook. La prioridad toma los

siguientes valores ordenados de mayor a menor prioridad (MH-must have, SH-should have y NH-nice to have):

Nombre	Pri	Descripción
Backlog management	MH	Como negocio e ingeniería, debemos seguir las prácticas definidas en el playbook de backlog management para definir correctamente la funcionalidad.
Build automatizada	MH	Como desarrollador, para ejecutar pruebas automáticas, necesito que la aplicación se haya compilado previamente de forma automática.
Pruebas configurables	SH	Como desarrollador, debo definir distintos tipos y conjuntos de pruebas para cada etapa del ciclo de vida, junto con su configuración asociada, la cual se debería versionar como código. Por ejemplo con cada commit ejecutar las pruebas smoke en navegadores y con backend virtualizado, con cada pull request ejecutar las pruebas de regresión, con cada deploy las pruebas post-deploy contra ese entorno y con cada release las pruebas de regresión utilizando emuladores/dispositivos tanto contra backend virtual como contra entornos.
Pruebas bajo demanda	SH	Como desarrollador, debo poder lanzar las pruebas cuando lo necesite sin tener que realizar cambios en la aplicación para ello. Además debo poder personalizar la configuración necesaria (tipo de prueba, entorno, juego de pruebas, etc).
Pruebas planificadas	SH	Como desarrollador, necesito configurar conjuntos de pruebas para ejecución periódica, con su respectiva configuración y acciones según los resultados obtenidos.
Trazabilidad e2e	SH	Como gerente, para evaluar rápidamente cambios, necesito trazabilidad entre pruebas, despliegues, modificaciones en el código y requerimientos.
Trazabilidad tests-defectos	MH	Como gerente, para evaluar la eficiencia de las pruebas necesito que cada cambio que incluya un correctivo tenga una marca que nos permita saber si el error asociado fué descubierto por una prueba automática.
Convenciones	SH	Como gerente, para medir y estandarizar procesos, debemos seguir convenciones en el uso de herramientas, como en la realización de cambios.
Repositorios funcionales	SH	Como Product Owner, para evitar cambios incorrectos en un repositorio, necesito ejecutar una versión funcional de la aplicación con cada uno de ellos.
Código-pruebas juntos	MH	Como Product Owner, es esencial prevenir modificaciones indebidas en un repositorio. Para ello, necesito asegurarme de que los cambios incluyan de manera integral las actualizaciones pertinentes tanto en el código como en las pruebas relacionadas.
Soporte pruebas	MH	Como desarrollador, para realizar pruebas, necesito que la arquitectura admita pruebas y tanto la arquitectura como la aplicación estén diseñadas para ello.
Soporte IDE	SH	Como desarrollador, para guiar mi desarrollo en base a las pruebas (TDD) necesito poder abrir la aplicación con todas sus dependencias en un IDE para así poder moldear el código y las pruebas según avanza de una manera eficiente (e.g refactorizando el código).
Entornos y pruebas locales	MH	Como desarrollador, para detectar los errores lo antes posible y realizar cambios en las pruebas, necesito ejecutar la aplicación en mi entorno local y lanzar las pruebas contra ella.
Pruebas pre-deploy	MH	Como equipo, debemos evitar desplegar cambios que rompan la aplicación en entornos compartidos y afecten a múltiples personas.
Enforcement merge	MH	Como equipo, debemos evitar integrar cambios en ramas compartidas que rompan la aplicación y afecten a múltiples personas.
Versionado SUT	SH	Como release manager, para validar cambios correctamente, necesito saber si el código/binario que se promocionará es idéntico al validado con las pruebas.

Pruebas impacto	NH	Como equipo, para reducir tiempos de ejecución y revisión de pruebas, necesitamos mecanismos para seleccionar las pruebas basadas en los cambios realizados.
InnerSource repos	NH	Como equipo, necesitamos proponer pruebas que nos gustaría que las aplicaciones dependientes ejecuten antes de liberar una nueva versión.
InnerSource pipelines	NH	Como equipo, para ajustar pruebas a nuestras necesidades, necesitamos adaptar o proponer cambios en el pipeline de construcción, pruebas y despliegues.
Pruebas pre-deploy efímeros/aislados	SH	Como equipo, necesitamos ejecutar la nueva versión de la aplicación en un entorno aislado para validar su funcionamiento antes de implantarla en un entorno compartido.
Pruebas pre-deploy virtualizadas	SH	Como equipo, para aplicar pruebas integradas en aplicaciones con dependencias inestables, necesitamos versiones funcionales de las mismas siempre disponibles.
Pruebas post-deploy/compartidos	MH	Como equipo, con cada implantación de cambios, ya sea bajo demanda, de forma automática con cada implantación, o de forma planificada, necesitamos validar que la aplicación funcione correctamente con todas sus dependencias y datos reales.
Saltar controles seguridad	MH	Como equipo, para automatizar pruebas como las integradas o de extremo a extremo, necesitamos poder saltar o automatizar de manera programática las medidas de seguridad en las operativas.
Rollback automático	SH	Como equipo, para minimizar la indisponibilidad de entornos, deseamos contar con un mecanismo de rollback para revertir cambios que hayan afectado negativamente la aplicación.
Enforcement despliegues	SH	Como equipo, para asegurar la máxima calidad en las implantaciones, necesitamos garantizar que la versión a desplegar haya pasado las pruebas obligatorias.
Versionado test-app-entornos	SH	Como Environment Cop, para garantizar la fiabilidad de las pruebas, requiero mecanismos para usar la versión correcta de las pruebas para validar la aplicación desplegada en un entorno específico.
Cambios entre pruebas	NH	Como equipo, para realizar triage efectivo de pruebas fallidas, necesitamos visualizar rápidamente todos los cambios desde la última ejecución exitosa de pruebas.
Canaries/Feature Toggles	NH	Como equipo, para llevar a cabo pruebas alfa o beta, necesitamos habilitar pruebas en producción para un grupo reducido de usuarios.
Interceptación sistemas externos	NH	Como equipo, para validar la correcta comunicación con terceros, como el envío de notificaciones push, requerimos mecanismos para interceptar peticiones y respuestas.
Trazas sistemas	NH	Como equipo, para realizar un triage efectivo de pruebas fallidas, necesitamos ver trazas de la aplicación en sus distintas capas durante la ejecución específica de cada prueba fallida.
Monitorización y alertas	NH	Como equipo, para asumir la responsabilidad de la calidad, necesitamos sistemas de monitoreo y alertas para cada entorno.
Carga de datos periódicas	MH	Como equipo, para probar funcionalmente la aplicación, necesitamos contar con los datos de prueba necesarios en las aplicaciones en todo momento. (e.g descargando y cargando diferentes usuarios con diferente frecuencia; desde minutos para algunos de ellos hasta semanas para otros)
Buscador de datos	MH	Como equipo, para probar funcionalmente la aplicación, requerimos algún mecanismo para localizar los datos necesarios.
Carga de datos bajo demanda	SH	Como equipo, para reproducir fácilmente problemas en producción o preparar los datos de prueba antes de su ejecución, necesitamos mecanismos para descargar y/o cargar usuarios específicos a demanda.
Reserva de datos	SH	Como equipo, para evitar conflictos entre equipos y aplicaciones, necesitamos un mecanismo para prevenir el uso simultáneo de los mismos datos por diferentes actores.

Cuadro 0.1 - Precondiciones y prioridad

Adicionalmente a lo anterior, se presenta el siguiente cuadro que recoge todas las prácticas y sub-prácticas del playbook y la prioridad necesaria de sus pre-condiciones:

	Prácticas																
Precondiciones	2.1.1	2.2.1	2.3.1	2.3.2	2.3.3	2.4	2.4.1	2.4.2	2.4.3	2.4.4	2.4.5	2.4.6	2.4.7	2.4.8	2.4.9	2.5.1	2.6.1
Backlog management	MH	MH											MH				MH
Build automatizada	SH	SH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	MH	SH	MH	MH
Pruebas configurables	NH	SH	SH	MH	MH	SH		SH	SH	SH	SH	SH	SH	SH	SH	SH	SH
Pruebas bajo demanda	SH	MH	SH	SH	SH	SH		SH	SH	SH	SH	SH	SH	SH	SH	SH	SH
Pruebas planificadas	NH	SH	SH	SH	SH	SH		SH	SH	SH	SH	SH	SH	SH	MH	MH	MH
Trazabilidad e2e	SH	SH	SH	SH	SH	SH		SH	SH	SH	SH	SH	MH	SH	SH	SH	SH
Trazabilidad tests-defectos	MH	MH	MH	MH	MH	MH		MH	MH	MH	MH	MH	MH	MH	MH	MH	MH
Convenciones	SH	MH	SH	SH	SH	MH		SH	SH	SH	SH	SH	SH	SH	SH	SH	SH
Repositorios funcionales	SH	MH	SH	SH	SH	MH		SH	SH	SH	SH	SH	SH	SH	NH	SH	SH
Código-pruebas juntos	SH	SH	MH	MH	MH	MH		MH	MH	MH	MH	MH	MH	MH	SH	SH	MH
Soporte pruebas	SH	SH	MH	MH	MH	MH		MH	MH	MH	MH	MH	MH	MH	MH	SH	MH
Soporte IDE	SH	SH	MH	MH	MH	MH	SH	MH	SH	SH	MH	SH	SH	SH	SH	SH	SH
Entornos y pruebas locales	SH	SH	MH	MH	MH	MH		MH	MH	MH	MH	MH	MH	MH	SH	SH	MH
Pruebas pre-deploy	SH	SH	MH	MH	MH	MH		MH	MH	MH	MH	MH	MH	MH		SH	MH
Enforcement merge	SH	SH	MH	MH	MH	MH		MH	MH	MH	MH	MH	MH	MH		SH	MH
Versionado SUT	NH	NH	NH	NH	SH	SH		SH	SH	SH	SH	SH	SH	SH	SH	SH	SH
Pruebas impacto		NH	NH	NH	NH	NH		NH	NH	NH	NH	NH	NH	NH		NH	NH
InnerSource repos		NH	NH	SH	SH	NH	NH	NH	NH	NH	NH	NH	NH	NH		NH	NH
InnerSource pipelines		NH	NH	SH	SH	NH	NH	NH	NH	NH	NH	NH	NH	NH		NH	NH
Pruebas pre-deploy efímeros/aislados		NH	SH	SH	SH	SH			MH	SH		SH	SH	MH		SH	MH
Pruebas pre-deploy virtualizadas		NH	SH	SH	SH	SH			MH	SH		SH	SH	MH		SH	MH
Pruebas post-deploy/compartidos	SH	SH	SH	SH	SH	SH			MH	MH		MH	MH	SH	MH	MH	MH
Saltar controles seguridad	NH	NH	SH	SH	SH	SH			MH	MH		MH	MH	SH	MH	MH	MH
Rollback automático		NH	SH	SH	SH	SH			SH	SH		SH	SH	SH	SH	MH	MH
Enforcement despliegues		NH	SH	SH	SH	SH			SH	SH		MH	MH	SH	SH	SH	MH
Versionado test-app-entornos	NH	NH	SH	SH	SH	SH			SH	SH		SH	SH	SH	SH	SH	SH
Cambios entre pruebas		NH	NH	NH	NH	NH			NH	NH		NH	NH	NH	SH	SH	NH
Canaries/Feature Toggles		NH	NH	NH	NH	NH			NH	NH		NH	NH	NH	NH	NH	NH
Intercepción sistemas externos		NH	NH	NH	NH	NH			NH	NH		NH	NH	NH		NH	NH
Trazas sistemas	NH	SH	NH	NH	NH	NH			NH	NH		NH	NH	NH	NH	SH	SH
Monitorización y alertas		MH	NH	NH	NH	NH			NH	NH		NH	NH	NH	NH	SH	SH

Carga de datos periódicas	SH	SH	MH	MH	MH	MH			MH	MH		MH	MH	MH	SH	MH	MH
Buscador de datos	SH	SH	MH	MH	MH	MH			MH	MH		MH	MH	MH	SH	MH	MH
Carga de datos bajo demanda	SH	SH	SH	SH	SH	SH			SH	SH		SH	SH	SH	SH	SH	SH
Reserva de datos	NH	NH	SH	SH	SH	SH			SH	SH		SH	SH	SH	SH	SH	SH
Usuarios productivos de pruebas	NH	NH	SH	SH	SH	SH			SH	SH		SH	SH	SH	SH	SH	SH

Cuadro 0.2 - Prioridad de las precondiciones por práctica.

Herramientas

En este apartado se recogen los requisitos que deben tener las herramientas de testing y el catálogo de herramientas identificadas en el Banco para las distintas pruebas previamente mencionadas.

El equipo QE Global debe proporcionar productos de prueba consumibles fáciles de usar e integrar por los equipos de productos de software, facilitando las prácticas de pruebas efectivas en el Banco y la colaboración entre los desarrolladores, los equipos y las geografías. Las herramientas deberían tener las siguientes características:

- **Documentación bien definida** debería estar disponible para descubrir fácilmente las herramientas de prueba en un mercado de desarrolladores *online* centralizado. Entre la documentación debería incluirse una guía de inicio completa, tutoriales prácticos, documentación de API, entornos de demostración con casos de uso de ejemplo y contactos y canales de asistencia.
- **Instanciación independiente** debería ser posible, p. ej., rotación de instancias, generación de API, acceso GIT, alto grado de configurabilidad.
- **Transparencia y autoservicio** en las operaciones para automatizar las tareas operativas cotidianas a través de API, p. ej., iniciar/detener ejecuciones de pruebas, configurar pruebas.
- **Un producto global** implementado en una base de código global de forma agnóstica para el usuario y ampliable para satisfacer la demanda global en el Banco.
- **Abierto a la contribución** con pista de auditoría de los cambios para producción a partir del *feedback* de los usuarios y base de código abierto para las solicitudes de contribución de todos los equipos del Banco.

Objetivo de la prueba	Tipo de prueba	Herramientas usadas en el Banco
Fiabilidad	Pruebas estáticas	Sonar
	Pruebas unitarias	Depende de la tecnología. Mocha, JUnit, unittest

		(Python Unit Testing Framework), Scala Test, WCT, mockito, vBank y similares
	Pruebas de integración	Depende de la tecnología: Mocha, JUnit, wdio/ acis , Cucumber, BackendTesting , Data-manager , galatea (saucelabs, browserstack y similares)
	Pruebas E2E/IU (incluidos smoke tests)	wdio/ acis , Cucumber, galatea (saucelabs, browserstack y similares)
	Pruebas de aceptación	XRay y el resto de herramientas para la automatización de pruebas
Seguridad	SAST	Chimera (que internamente usa las herramientas comerciales Kiuwan y CheckMarx). Para entorno local, a definir por CISO GSD.
	DAST	Herramienta(s) a definir por CISO GSD.
	IAST	Herramienta(s) a definir por CISO GSD.
	RASP	Herramienta(s) a definir por CISO GSD.
	Pruebas de intrusión / Hacking ético	Las herramientas utilizadas en las pruebas de intrusión varían mucho. Generalmente, se utiliza una combinación de herramientas automáticas y manuales.
	SCA (Software Composition Analysis)	Chimera (que internamente usa la herramienta open-source Dependency Track). Para entorno local, a definir por CISO GSD.
	Detección de secretos	Chimera (que internamente usa la herramienta in-house Secrets).
	Imágenes (seguridad en contenedores)	Chimera (que internamente usa la herramienta comercial Prisma Cloud).
Resiliencia (rendimiento)	Pruebas de carga	neoload, loadrunner, jmeter, DQ Portal, PMaaS
	Pruebas de resistencia	Herramienta(s) a definir
	Ingeniería de caos	Herramienta(s) a definir
Experiencia de	Pruebas de	axe

usuario	accesibilidad	
	Pruebas de compatibilidad	galatea (saucelabs, browserstack y similares)
	Gestión backlog (tests)	Jira

Objetivo de la prueba	Herramientas usadas en el Banco
Reporting	Test-Manager , DevOps Metrics
Data Management	Data-Manager

Cuadro 0.3 - Herramientas de prueba en el Banco.

GLOSARIO

Equipo: Conjunto de personas trabajando sobre un mismo backlog de tareas asociado a un producto de software.

Servicio: Productos de software (e.g., aplicación web, aplicación móvil, librerías de transacciones).