Modelo de Mejora para Pruebas Continuas

Tesis presentada para obtener el grado de Doctor en Ciencias Informáticas

Autor

Maximiliano Agustin Mascheroni

Directores

Dr. Emanuel Irrazábal[1]

Dr. Gustavo Rossi[2]

La Plata, marzo de 2021

Facultad de Informática – Universidad Nacional de La Plata

[1] Facultad de Ciencias Exactas y Naturales y Agrimensura. Universidad Nacional del Nordeste
[2] Facultad de Informática. Universidad Nacional de La Plata.

Agradecimientos

A mi familia que siempre ha aportado en lo que soy, y por lo tanto, en lo que hago. A Marie, por apoyarme a lo largo de todo este proceso, y a su familia por acompañar. Además, a todos mis amigos que siempre estuvieron alentando: Nico, Emilio, Luis, Juan, Marcos, Guille, Fede, Seba y Christian.

A Emanuel y Gustavo, por la dirección de este trabajo y de mi carrera profesional. A los docentes del equipo de investigación de la Universidad Nacional del Nordeste, en especial a Laly Dapozo, Cristina Greiner y Raquel Petris. A Viviana Godoy por el apoyo y a mis compañeros investigadores.

A la empresa TradeHelm, sobre todo a Juli Paniego y Mati Mazzuchelli, por la consideración y el respaldo desde el inicio del Doctorado. A colegas como Karen Bark, Drew Griffin y Guy Finkill. También a la empresa Crowdar por el apoyo, en especial a Javier Re, Cecilia Benassi, Pablo Zucarino, y Juan Manuel Carames.

A la Facultad de Informática de la Universidad Nacional de La Plata, y en general, a todos los que lean este trabajo.

RESUMEN

La Entrega Continua es una práctica donde se desarrolla software de calidad de un modo en el que puede ser lanzado a producción en cualquier momento. Sin embargo, como parte de este trabajo de investigación se realizaron una revisión sistemática de la literatura y una encuesta, las cuales reportan que tanto la literatura académica como la industria todavía encuentran problemas relacionados con el proceso de pruebas al usar prácticas como Entrega Continua o Despliegue Continuo. De este modo, se propone el Modelo de Mejora para Pruebas Continuas como una solución a los problemas de pruebas en los entornos de desarrollo continuo. El mismo recopila propuestas y enfoques de diferentes autores que son presentados como buenas prácticas, agrupadas por tipos de pruebas y divididos en cuatro niveles. Estos niveles indican una jerarquía de mejora y un camino evolutivo en la implementación de las Pruebas Continuas. Además, una herramienta llamada EvalCTIM fue desarrollada para guiar la evaluación del proceso de pruebas utilizando el modelo propuesto. Finalmente, para validar el modelo, se empleó el método de Investigación-Acción mediante una evaluación teórica interpretativa seguida de estudios de casos llevados a cabo en proyectos de desarrollo de software reales. Los resultados demuestran que el modelo se puede utilizar como una solución para implementar las Pruebas Continuas de forma gradual en empresas con entornos de desarrollo continuo.

ABSTRACT

Continuous Delivery is a practice where high-quality software is built in a way that it can be released into production at any time. However, a systematic literature review and a survey performed as part of this research report that both the literature and the industry are still facing problems related to testing using practices like Continuous Delivery or Continuous Development. Thus, we propose Continuous Testing Improvement Model as a solution to the testing problems in continuous software development environments. It brings together proposals and approaches from different authors which are presented as good practices grouped by type of tests and divided into four levels. These levels indicate an improvement hierarchy and an evolutionary path in the implementation of Continuous Testing. Also, an application called EvalCTIM was developed to support the appraisal of a testing process using the proposed model. Finally, to validate the model, an action-research methodology was employed through an interpretive theoretical evaluation followed by case studies conducted in real software development projects. The results demonstrate that the model can be used as a solution for implementing Continuous Testing gradually at companies using continuous software development practices.

ÍNDICE

ÍNDICE DE FIGURAS

ÍNDICE DE TABLAS

ÍNDICE DE ALGORITMOS

ABREVIATURAS

API	Interfaz de programación de aplicaciones
BDD	Desarrollo guiado por comportamiento
CPLS	Capacidad, Rendimiento, Carga y Esfuerzo
CTIL	Nivel de mejora de Pruebas Continuas
CTIM	Modelo de mejora de Pruebas Continuas
DC	Despliegue Continuo
DSL	Desarrollo de software Lean
DP	Conducto de Despliegue
DTO	Objeto de transferencia de datos
E2E	De extremo a extremo
EC	Entrega Continua
GCR	Grupo Crítico de Referencia
GUI	Interfaz Gráfica de Usuario
IA	Investigación-Acción
IA-SI	Investigación-Acción en Sistemas de Información
IC	Integración Continua
IDE	Entorno de desarrollo integrado
IEEE	Instituto de Ingeniería Eléctrica y Electrónica
ISO	Organización Internacional de Normalización
ISTQB	Comité Internacional de Certificaciones de Pruebas de Software
OPS	Equipo de Operaciones e Infraesctructura
RIA	Aplicaciones de Internet Enriquecidas
RNF	Requerimientos no funcionales
PAC	Plan de acción correctivo
PC	Pruebas Continuas
SCV	Sistema de control de versiones
SO	Sistema Operativo
SWEBOK	Cuerpo de Conocimiento de la Ingeniería de Software
TaaS	Pruebas como un Servicio
TDD	Desarrollo guiado por pruebas
V&V	Verificación y validación

1. INTRODUCCIÓN

En primer lugar, se presenta la motivación de esta tesis (sección 1.1). Luego se enuncian los objetivos (sección 1.2), las áreas de conocimiento implicadas (sección 1.3), la metodología utilizada (sección 1.4), el alcance (sección 1.5) y la producción científica derivada de resultados parciales de la misma (sección 1.6). Por último, se describe la organización del documento (sección 1.7).

1.1. Motivación

Tradicionalmente, el software se construía utilizando metodologías secuenciales, donde una etapa no podía comenzar hasta que la anterior no haya terminado [1]. En este tipo de metodologías, la ejecución de pruebas es una de las últimas etapas haciendo que la detección de defectos en el sistema aumente los costos y los retrasos en los tiempos de entrega [2].

El desarrollo ágil de software ha incorporado una gran variedad de buenas prácticas y procesos para producir software con mayor velocidad y eficiencia [3]. Una de las más conocidas es la Integración Continua (IC), una práctica del desarrollo de software, en donde los programadores integran el código fuente con frecuencia y cada integración es verificada por un sistema que construye el código y lo prueba automáticamente [4]. Posterior a esta buena práctica, se han ido incorporando otras como la automatización de los despliegues al entorno de producción.

Por un lado, algunas compañías como Facebook [5] buscaban automatizar el proceso completo, desde la integración del código fuente, hasta el despliegue a producción; a ese primer enfoque se lo denominó Despliegue Continuo (DC) [6]. Por otro lado, otras empresas, decidieron automatizar todo el flujo de trabajo de tal manera que el despliegue a producción se realizaría de manera automática al “presionar un botón”; a este segundo enfoque se lo llamó Entrega Continua (EC) [7].

En este contexto, la calidad del software cumple un rol fundamental, ya que al automatizar los procesos de desarrollo, el tiempo para ejecutar los diferentes tipos y niveles de pruebas se reduce considerablemente [8]. Se han propuesto diferentes herramientas para hacer frente a estas problemáticas. Entre ellas se encuentran las pruebas automatizadas, que reducen el tiempo de ejecución y reemplazan la verificación manual que puede generar errores por tratarse de tareas realizadas por el humano. Para Entrega Continua, se propone un patrón llamado conducto

de despliegue[3] (DP) [7]. En Integración Continua, se recomienda automatizar las pruebas unitarias, de integración y de rendimiento e incluirlas en los scripts de construcción [9]. Estos scripts luego tienen que ser ejecutados de manera local por los desarrolladores y posteriormente en el servidor de Integración Continua, al introducirse los cambios en el repositorio de control de versiones. En Entrega Continua, se busca automatizar más tipos y niveles de pruebas [7] e introducirlos en el conducto de despliegue, dividiendo el flujo en diferentes etapas y aumentando la confiabilidad al avanzar en cada una, usualmente al costo de más tiempo de ejecución [10].

Sin embargo, según Duvall, Matyas y Glover [9] encontrar un equilibrio entre un tiempo de ejecución pequeño (velocidad) y un lote de pruebas que aseguren la calidad de cada entregable (exhaustividad) es una tarea difícil de llevar a cabo. Por un lado, si las pruebas son de extensa duración, se convierte en un proceso tedioso y generador de bloqueos en los desarrolladores, mientras esperan por retroalimentación. Por otro lado, si las pruebas no son lo suficientemente exhaustivas, se pueden introducir errores costosos o fallos críticos en producción.

En una serie de estudios realizados recientemente sobre Entrega Continua [11]–[14], se demostró que aún existen problemas en relación a las pruebas en proyectos de desarrollo que utilizan prácticas como IC, DC o EC. Los problemas más importantes reportados son: pruebas de integración, pruebas automatizadas que generan falsos positivos, pruebas automatizadas difíciles de mantener sobre una interfaz gráfica, tiempo de ejecución de las pruebas elevado y falta de cobertura de ciertos tipos pruebas en entornos continuos.

Por otro lado, la industria también ha reportado problemas relacionados al proceso de pruebas en entornos continuos [15]. Los problemas más críticos para la industria son las pruebas que requieren mucho tiempo de ejecución, las pruebas inestables, las pruebas no deterministas, las pruebas automatizadas de interfaz gráfica, las pruebas de Big Data y la falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en entornos de desarrollo continuo [15]. Esto demuestra que tanto la literatura académica como la industria se encuentran alineados.

Finalmente, tanto la industria del software [11] como los trabajos académicos [15] afirman que es necesario un modelo formal para la implementación de pruebas, al adoptar enfoques de desarrollo continuo. De acuerdo a Prusak [16], “la industria aún no ha cerrado el círculo cuando se trata de implementar un proceso completo de Entrega Continua”. Es por ello que para algunos autores, las Pruebas Continuas son el elemento faltante en el proceso de desarrollo continuo

[3] *Deployment Pipeline*

[16], [11], [17]. Como parte del proceso de Pruebas Continuas, se han propuesto gran cantidad de técnicas, métodos y herramientas para hacer frente a los problemas relacionados a las pruebas en entornos continuos [11], [12], [18]–[20]. Sin embargo, la implementación de este proceso en entornos continuos ha sido un desafío en la práctica [12]. Algunas organizaciones no han podido adoptar estos enfoques completamente [21] y otras han encontrado gran cantidad de obstáculos [13], [22], [23], [24], [25], [26], [27].

Es en este punto donde surge el Modelo de Mejora para Pruebas Continuas como una solución al problema de los procesos de pruebas en entornos continuos. En particular, este modelo ha sido desarrollado dentro de un proyecto de investigación en calidad de software con la colaboración de empresas nacionales e internacionales que desarrollan diferentes tipos de plataformas y sistemas.

Este modelo, reúne propuestas, técnicas, herramientas y enfoques de diferentes autores, que son presentadas como buenas prácticas agrupadas por tipo de pruebas (unitarias, funcionales, no funcionales, etc.) y divididas en niveles que indican una jerarquía de mejora y un camino por etapas para evaluar el progreso del proceso de pruebas de un proyecto de desarrollo en un entorno continuo. Asimismo, las organizaciones pueden validar el progreso en determinados aspectos de las Pruebas Continuas, mediante una auditoría realizada con el modelo. Para ello, también se presenta un soporte tecnológico para llevar a cabo las evaluaciones.

1.2. Hipótesis y Objetivos

La hipótesis planteada es la siguiente: tomando las nuevas propuestas, técnicas y herramientas que están surgiendo como parte del proceso de Pruebas Continuas, es factible la construcción de un modelo formado por buenas prácticas, agrupadas por etapas según los tipos de pruebas existentes y dividido en niveles de mejora para que una organización pueda medir el progreso en la implementación del proceso de Pruebas Continuas.

Se ha propuesto como objetivo principal de este trabajo de tesis, la construcción de un modelo, compuesto por distintas etapas y niveles de mejora, que permita la implementación de Pruebas Continuas y apoyen el desarrollo continuo de software.

1.2.1. Objetivos Específicos

A continuación se enumeran los objetivos específicos de esta tesis en base al objetivo general descrito anteriormente:

- **O1**. Identificar y relevar los problemas existentes relacionados a las pruebas de software en los procesos de desarrollo continuo, tanto en la literatura académica como en la industria.
- **O2**. Analizar los últimos avances, propuestas, herramientas, enfoques y soluciones existentes para los problemas detectados.
- **O3**. Construir un modelo de Pruebas Continuas, mediante la combinación de las técnicas y herramientas existentes, como así también de las diferentes propuestas que existen en la investigación académica.
- **O4**. Desarrollar una herramienta para evaluar el proceso de pruebas de una organización en base al modelo construido.
- **O5**. Implementar y validar el modelo en ambientes reales de desarrollo, que utilicen o busquen el enfoque de desarrollo continuo de software.

1.3. Áreas de Conocimiento Implicadas

Con el fin de aumentar el grado de estructuración de esta tesis, se toma como marco de referencia el documento Software Engineering Body of Knowledge (SWEBOK) [28], creado por la Software Engineering Coordinating Committee y promovido por la IEEE Computer Society. El documento significa una guía del conocimiento en el área de Ingeniería del Software, como un paso esencial hacia el desarrollo de la profesión, ya que representa un amplio consenso respecto a los contenidos de la disciplina y ordena todo el conocimiento existente en una ontología acordada por la IEEE. Su última versión ha sido liberada en el año 2013.

Según el SWEBOK, el conocimiento en Ingeniería del Software está organizado en 15 áreas de conocimiento:

- Requisitos de Software
- Diseño de Software
- Construcción de Software
- Pruebas de Software
- Mantenimiento de Software

- Gestión de la configuración
- Gestión de la Ingeniería de Software
- Proceso de Ingeniería de Software
- Herramientas y métodos de la Ingeniería de Software
- Calidad del Software
- Práctica Profesional de la Ingeniería de Software
- Economía de la Ingeniería de Software
- Fundamentos de Computación
- Fundamentos Matemáticos
- Fundamentos de Ingeniería

Tal como ilustra la Fig. 1, de acuerdo con el SWEBOK este trabajo se encuentra enmarcado dentro de las siguientes áreas de conocimiento:

- Área de “Software Testing” (Pruebas de Software): este trabajo se enmarca principalmente en esta área y todas sus subáreas: “Software Testing Fundamentals” (Fundamentos de las Pruebas de Software), “Test Levels” (Niveles de Pruebas), “Test Techniques” (Técnicas de Pruebas), “Test Related Measures” (Medidas relacionadas a las Pruebas), “Test Process” (Procesos de Pruebas).
- Área de “Software Configuration Management” (Gestión de la Configuración de Software): en segundo lugar, el trabajo se relaciona con las subáreas de “Management of the SCM Process” (Administración del Proceso de Gestión de la Configuración), “Software Configuration Identification” (Identificación de configuraciones del Software) y “Software Release Management and Delivery” (Gestión y Entrega de la Liberación del Software), que forman parte de esta área.
- Área de “Software Engineering Process” (Proceso de Ingeniería del Software): por otro lado, este trabajo también se enmarca en esta área, particularmente en las subáreas de “Process Definition” (Definición de Procesos), “Process Implementation and Change” (Implementación y Cambios en los Procesos), “Process Assessment” (Evaluación de los Procesos), “Process and Product Measurement” (Medición de los Procesos y Producto).
- Área de “Software Engineering Tools and Methods” (Herramientas y Métodos de Ingeniería de Software): ésta es otra área que forma parte del contexto de este trabajo.
- Área de “Software Quality”: finalmente, el trabajo se enmarca en esta área, dentro de las subáreas de “Software Quality Fundamentals” (Fundamentos de la Calidad del Software), “Software Quality Management Process” (Procesos de Gestión de la Calidad del

Software) y “Software Quality Practical Considerations” (Consideraciones Prácticas de la Calidad del Software).

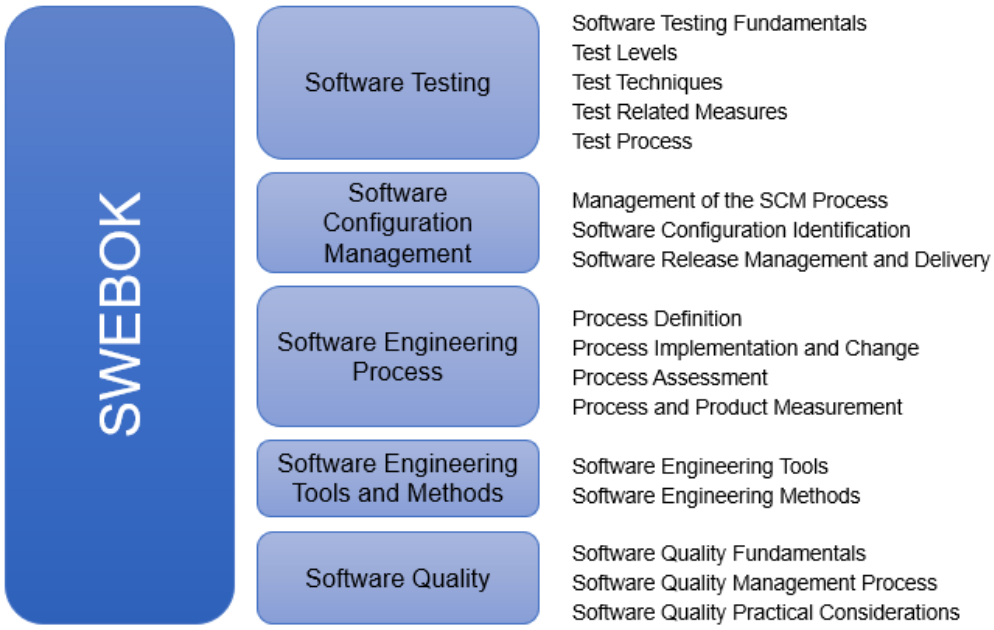


Fig. 1. Áreas y subáreas de SWEBOK donde se enmarca el trabajo.

1.4. Metodología

La diferente naturaleza de las Ingenierías, respecto al resto de las disciplinas empíricas y formales, dificulta la aplicación directa de los métodos de investigación clásicos y en particular, a la investigación en ingeniería de software.

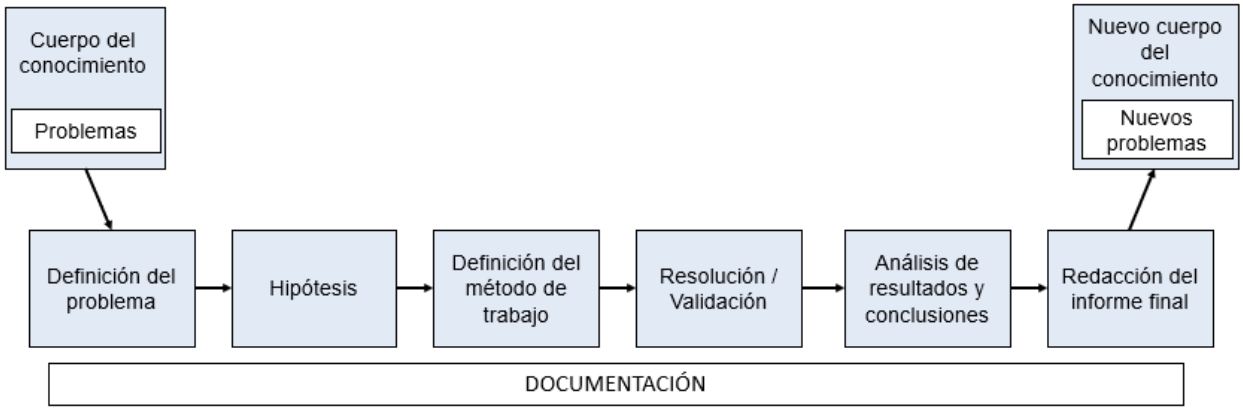


Fig. 2. Método de Investigación.

El método de investigación que se siguió en esta tesis ha sido una adaptación del propuesto por Marcos y Marcos [29], [30] para la investigación en Ingeniería del Software. Dicho método se basa en el hipotético-deductivo presentado por Bunge [31] y se compone de una serie de pasos que se muestran en la Fig. 2.

La recopilación de información relacionada al estado del arte se realizó mediante revisiones sistemáticas de la literatura y encuestas. El diseño, desarrollo e implementación del modelo y de la herramienta, se llevó a cabo utilizando metodologías ágiles. Las mismas, al ser iterativas e incrementales, permitieron someter a las etapas de diseño y desarrollo, a procesos de mejora continua según los resultados parciales que se iban obteniendo en los ciclos de implementación y validación que se describen más adelante. Este tipo de metodologías han demostrado ser eficaces en la organización y el desarrollo de Tesis Doctorales [32].

1.4.1. Documentación del estado del arte y la descripción del problema

Para el desarrollo de esta etapa se ha utilizado los métodos de revisión sistemática de la literatura [33] y encuesta [34].

El método de revisión sistemáticas se utiliza para identificar, evaluar e interpretar toda la información relacionada a un tema de investigación en particular, de un modo sistemático y replicable. Surge de la investigación en el campo de la medicina y se ha convertido en una metodología confiable, rigurosa y auditable. La aplicación de las revisiones sistemáticas en el ámbito de la Ingeniería del Software permite dar un valor científico a la revisión de la literatura que se hace, definir una estrategia de búsqueda de los trabajos a evaluar y obtener finalmente una hipótesis a favor o en contra de dicha literatura.

Para el desarrollo de este trabajo de investigación, se ha tomado como referencia la adaptación del método de revisiones sistemáticas para Ingeniería del Software presentado en [35]. En dicho documento se propone una nueva aproximación en la cual el proceso de revisión sistemática está compuesto por tres grandes fases: planificación, ejecución y análisis de los resultados obtenidos, tal como se muestra en la Fig. 3.

Fig. 3. Proceso de método de revisiones sistemáticas.

En la fase de **planificación** se identifican claramente los objetivos de la investigación y se define el protocolo de revisión a utilizar. Además, se deben establecer los criterios de inclusión y exclusión que se seguirán para determinar los artículos a seleccionar y sus fuentes.

En la fase de **ejecución** se realiza lo planificado en la etapa anterior, determinando en principio el conjunto de estudios o documentos a revisar. Estos artículos se seleccionan a través de la evaluación, para cada uno de ellos, de los criterios de inclusión y exclusión elegidos en la etapa de planificación. El paso final de esta fase es la extracción de la información más relevante por cada estudio.

Finalmente, en la etapa de **análisis de resultados**, se sintetiza y evalúa la información extraída de cada estudio.

Existe además una fase de **resguardo de resultados,** que se realiza durante todo el proceso de la revisión sistemática, ya que a medida que se llevan a cabo cada una de las etapas, el resultado de las mismas debe ser registrado.

Como se puede ver en la Fig. 3, existen puntos de verificación a lo largo del proceso. El primer punto garantiza que la planificación realizada es la adecuada; para ello, se debe evaluar el protocolo definido y, si hubiera algún problema o incongruencia, se debería volver a la fase anterior. El segundo punto de verificación debe realizarse una vez acabada la fase de ejecución; de la misma manera que en el punto anterior, si hubiera algún error en los resultados de esta etapa se debería ejecutar de nuevo la selección y/o extracción de información. En la sección 3.2 se presentará la revisión sistemática realizada conforme al proceso que se acaba de describir.

Por otro lado, también se ha utilizado el método de encuesta. De acuerdo a Genero Bocco et al. [36], las encuestas son quizás el método de investigación más utilizado por todo el mundo. Una encuesta es un método de investigación empírico utilizado para recopilar información de o sobre personas para describir, comparar o explicar su conocimiento, sus actitudes o su comportamiento [37]. También se utilizan para la descripción de las características de métodos o herramientas.

En la mayoría de los casos, los datos relativos a la encuesta provendrán de cuestionarios. Sin embargo, una encuesta no consiste en un cuestionario. Una encuesta es un proceso complejo formado por una serie de actividades bien definidas [38], que se presentan en la Fig. 4.

La encuesta realizada para la obtención de los resultados de este trabajo se presentará en la sección 3.4

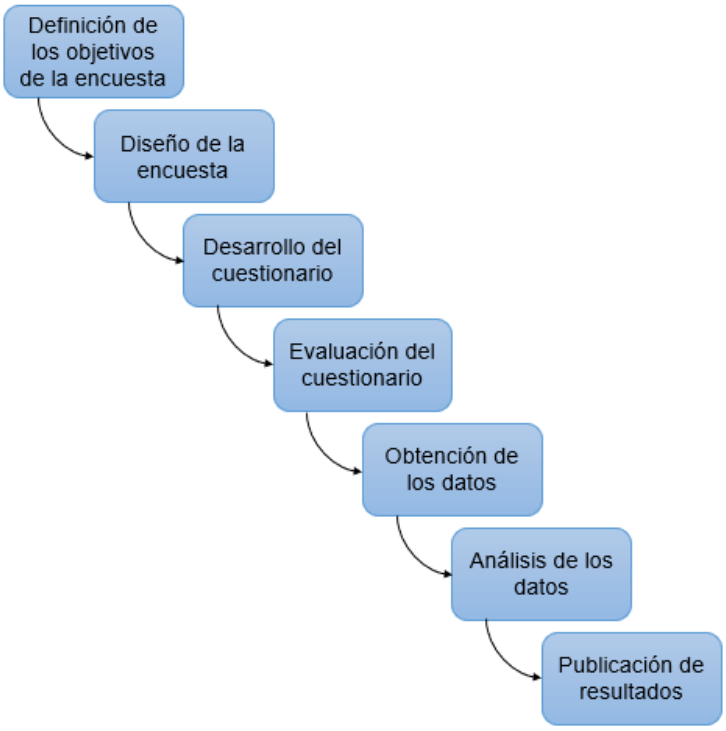


Fig. 4. Etapas en el método de encuesta.

1.4.2. Desarrollo, resolución y validación de los resultados obtenidos

Para esta etapa se utiliza el método de Investigación-Acción (IA). El término "Investigación-Acción" proviene del autor Kart Lewin [39] que lo utilizaba para describir una forma de investigación que podía enlazar el enfoque experimental de las ciencias sociales con programas de acción social que respondieran a los problemas sociales principales de entonces. Mediante la IA, Lewin argumentaba que se podían lograr de forma simultánea avances teóricos y cambios sociales. A pesar de que la primera propuesta de IA fue introducida en 1985 [40], en los últimos años ha obtenido una gran atención y aceptación por parte de la comunidad investigadora en Sistemas de Información [41], [42].

Investigación-Acción no se refiere a un método de investigación concreto, sino a una clase de métodos que tienen en común las siguientes características:

- Orientación a la acción y al cambio
- Focalización en un problema
- Un modelo de proceso "orgánico" que engloba etapas sistemáticas y algunas veces iterativas y
- Colaboración entre los participantes.

Se define este método de investigación como “la forma que tienen los grupos de personas de preparar las condiciones necesarias para aprender de sus propias experiencias y hacer estas experiencias accesibles a otros” [43].

La IA tiene una doble finalidad: generar un beneficio al “cliente” de la investigación y, al mismo tiempo, generar “conocimiento de investigación” relevante [44]. Por tanto, es una forma de investigar de carácter colaborativo, que busca unir teoría y práctica entre investigadores y profesionales mediante un proceso de naturaleza cíclica. La IA está orientada a la producción de nuevo conocimiento útil en la práctica, que se obtiene mediante el cambio y/o búsqueda de soluciones a situaciones reales que le ocurren a un grupo de profesionales [42]. Esto se consigue gracias a la intervención de un investigador en la realidad del mencionado grupo. Los resultados de esta experiencia deben ser beneficiosos tanto para el investigador como para los profesionales.

En el campo de los Sistemas de Información, el cliente de una investigación es normalmente una organización a la cual el investigador suministra servicios tales como consultoría, ayuda para cambiar o desarrollo de software, a cambio de tener acceso a datos de interés para la investigación y, en muchos casos, de recibir financiación [44]. En cualquier caso, el investigador que utiliza IA en Sistemas de Información (IA-SI) sirve a dos entidades diferentes: el cliente de la investigación y la comunidad científica de Sistemas de Información. Las necesidades de ambos suelen ser muy diferentes y, a veces, opuestas entre sí. Intentar satisfacer ambas demandas es el principal desafío que todos los investigadores de IA-SI tienen que enfrentar.

En este caso se aplicará el método de Investigación-Acción como resultado de la colaboración entre la actividad investigadora del doctorando y su aplicación en proyectos de desarrollo de software de empresas internacionales, como TradeHelm, Google, Globant, Cision, National Geographic, Almundo, Aerolíneas Latam, CME Group, Southwest Airlines, Vontobel, Philips, Classified Ventures, Grupo Clarin, Thomson Reuters, Kapsh, Highmetric, entre otras.

Los roles identificados en Investigación-Acción [45] son el investigador, el objeto investigado, el grupo crítico de referencia (GCR) y los beneficiarios. La Tabla 1 muestra los roles asociados a este trabajo. El rol de investigador lo tiene el doctorando junto con un grupo de expertos que son mencionados en la sección 6. El objeto de investigación es el modelo descrito en el apartado 4, como resultado del presente trabajo. El grupo crítico de referencia está compuesto por los profesionales de desarrollo software de las empresas mencionadas.

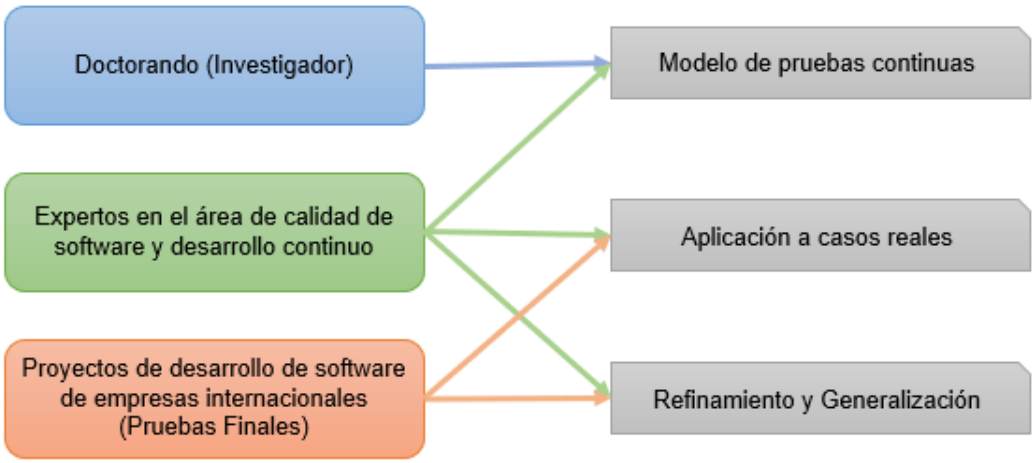


Fig. 5. Fases del método de Investigación-Acción utilizado.

La Fig. 5 muestra la relación de propuesta, validación y refinamiento de propuestas, involucrando al investigador, el grupo crítico de referencia y los beneficiarios. El proceso consta de una etapa de evaluación e implementación de buenas prácticas que conforman el modelo, la cual será llevada a cabo por el doctorando (investigador) y expertos en el campo de investigación. Posteriormente, se contrasta el modelo a través de las validaciones realizadas con los datos de proyectos reales suministrados por las empresas. Finalmente, se refinan las propuestas iniciales utilizando las mediciones y la experiencia de uso.

Tabla 1. Roles identificados en la Investigación-Acción.

Rol	Descripción	Rol identificado en este trabajo
Investigador	El individuo o grupo que lleva a cabo de forma activa el proceso investigador.	El doctorando.
Objeto Investigado	El problema a resolver.	El modelo de mejora para Pruebas Continuas.
Grupo Crítico de Referencia (GCR)	Aquel para quien se investiga en el sentido de que tiene un problema que necesita ser resuelto y que también participa en el proceso de investigación.	Expertos en calidad de software y desarrollo continuo.
Beneficiario	Aquel para quien se investiga en el sentido de que puede beneficiarse del resultado de la investigación, aunque no participa directamente en el proceso.	Cualquier proyecto de desarrollo de software que utilice prácticas del desarrollo continuo.

1.5. Alcance

En la actualidad, se han reportado un gran número de problemas y soluciones relacionados a las Pruebas Continuas. Por un lado, cada problema está asociado a líneas de investigación, donde se lo estudia, analiza y, por otro lado, se proponen soluciones para enfrentarlo.

El modelo propuesto en este trabajo recopila las diferentes herramientas y soluciones propuestas por diversos investigadores (incluido el doctorando) y empresas, verificadas en proyectos reales de desarrollo de software. Dentro del modelo, estas soluciones han sido distribuidas en diferentes niveles de mejora relacionados directamente con estándares y modelos de madurez para organizaciones de desarrollo de software, que a su vez están divididos por etapas representativas de niveles de pruebas (unitarias, funcionales, no funcionales, etc.)

La validación del modelo demuestra que el mismo puede ser utilizado por organizaciones con diferentes niveles de madurez, independientemente del tipo de plataforma que desarrollen (web, aplicaciones móviles y de escritorio). Además, si bien el modelo se centra en aquellos proyectos que utilicen el enfoque de desarrollo continuo de software, o que buscan alcanzarlo, también se lo puede implementar en equipos de desarrollo con otros enfoques.

Sin embargo, uno de los problemas más actuales en el desarrollo continuo relacionado al proceso de pruebas, es Big Data. Como se presentará más adelante, aún no hay evidencia en la literatura sobre la existencia de soluciones totales para este tópico. Por esta razón, el modelo propuesto no contempla los problemas de estas características, los cuáles serán parte de los trabajos futuros.

Del mismo modo, el modelo tampoco significa una solución para los problemas relacionados con el proceso de pruebas (responsabilidades, planificación, estimación, estrategia, comunicación, colaboración en el equipo, guías, etc.)

1.6. Lista de publicaciones derivada de resultados durante el desarrollo de la tesis

Durante el desarrollo de esta tesis se han comunicado resultados parciales a través de diversas publicaciones que se detallan a continuación.

- Mascheroni, M.A.; Greiner, C.L.; Petris, R.H., Dapozo, G.N.; & Estayno, M. G. (abril, 2012). “Calidad de software e ingeniería de usabilidad”. XIV Workshop de Investigadores en Ciencias de la Computación (WICC 2012), pp. 656-660

- Mascheroni, M.A.; Greiner, C.L; & Petris, R.H. (junio, 2012). “Técnicas de usabilidad. Estudio exploratorio sobre su incorporación en los procesos de desarrollo de software en pymes locales”. II Jornadas de Investigación en Ingeniería del NEA y países limítrofes.

- Mascheroni, M.A.; Greiner, C.L.; Dapozo, G.N.; & Estayno, M.G. (octubre, 2012). “Herramienta para automatizar la evaluación de la usabilidad en productos software”. XVIII Congreso Argentino de Ciencias de la Computación (CACIC 2012), pp. 947-956.

- Mascheroni, M.A.; Greiner, C.L.; Dapozo, G.N.; & Estayno, M.G. (abril, 2013). “Automatización de la evaluación de la usabilidad del software”. XV Workshop de Investigadores en Ciencias de la Computación (WICC 2013), pp. 572-575.

- Mascheroni, M.A.; Greiner, C.L.; Dapozo, G.N.; & Estayno, M.G. (junio, 2013). “Ingeniería de Usabilidad. Una Propuesta Tecnológica para Contribuir a la Evaluación de la Usabilidad del Software”. Revista Latinoamericana de Ingeniería de Software, Vol. 1, N° 4, pp. 125-134.

- Dapozo, G.N.; Greiner, C.L.; Irrazabal, E.; Medina, Y.; Ferraro, M.D.L.A.; Lencina, A.; Chiapello, J.; & Mascheroni, M.A. (abril, 2016). “Métodos y herramientas de estimación, gestión cuantitativa de proyectos, trazabilidad de requerimientos y entrega continúa, orientados a la mejora de la calidad del software”. XVIII Workshop de Investigadores en Ciencias de la Computación (WICC 2016)., pp. 651-656.

- Mascheroni, M.A.; Cogliolo, M.K.; & Irrazábal, E. (septiembre, 2016). “Automatización de pruebas de compatibilidad web en un entorno de desarrollo continuo de software”. Simposio Argentino de Ingeniería de Software (ASSE 2016) – 45 Jornadas Argentinas de Informática (JAIIO), pp. 51-63.

- Mascheroni, M.A.; & Irrazábal, E. (octubre, 2016). “Framework para la creación y ejecución de pruebas automatizadas sobre servicios REST”. XXII Congreso Argentino de Ciencias de la Computación (CACIC 2016), pp. 495-504.

- Dapozo, G.N.; Greiner, C.L.; Irrazábal, E.; Medina, Y.; Ferraro, M.A.; & Mascheroni, M.A. (abril, 2017). “Gestión Cuantitativa de Proyectos y Entrega Continúa en Entornos Ágiles”. XIX Workshop de Investigadores en Ciencias de la Computación (WICC 2017), pp. 525-531.
- Mascheroni, M.A.; & Irrazábal, E. (septiembre, 2017). “A Design Pattern Approach for RESTful tests: A case study”. 12th Colombian Conference on Computing (CCC 12), Cali, Colombia.
- Mascheroni, M.A.; Cogliolo, M.K.; & Irrazábal, E. (septiembre, 2017). “Automatic detection of Web Incompatibilities using Digital Image Processing”. Electronic Journal of Informatics and Operations Research, Special Issue dedicated to JAIIO 2016. Vol. 16, N° 1, pp. 29-45.
- Sabaren, L.; Mascheroni M.A.; & Irrazábal, E. (octubre, 2017). “Una revisión sistemática de la literatura en pruebas de compatibilidad web”. XXIII Congreso Argentino de Ciencias de la Computación (CACIC 2017), pp. 812-821.
- Sabaren, L.; Mascheroni, M.A.; & Irrazábal, E. (abril, 2018). “A Systematic Literature Review in Cross-browser Testing”. Journal of Computer Science and Technology, Vol 18, N° 1, e03.
- Irrazábal, E.; Mascheroni, M.A.; Alonso, J.M.; Vier, S., Pereyra Coimbra, R.; Del Yesso, A. (abril, 2018). “Ingeniería de software para sistemas embebidos, requisitos y entrega continua”. XX Workshop de Investigadores en Ciencias de la Computación (WICC 2018), pp. 553-557.
- Mascheroni, M.A.; & Irrazábal, E. (2018). “A Design Pattern Approach for RESTful tests: A case study”. Obras colectivas en ciencias de la computación, pp. 257-268. Cali, Colombia.
- Mascheroni, M.A.; & Irrazábal, E. (septiembre, 2018). “Continuous Testing and solutions for testing problems in Continuous Delivery: A Systematic Literature Review”. Revista Computación y Sistemas, Vol. 22, N°3, pp. 1009–1038.
- Mascheroni, M.A.; & Irrazábal, E. (octubre 2018). “Identifying key success factors in stopping flaky tests in automated REST service testing”. Journal of Computer Science & Technology, Vol 18, N° 2, e16.

- Irrazábal, E.; Mascheroni, M.A.; Sambrana, I.; Alonso J.M.; Acevedo J.; & Lezcano A. (abril, 2019). “Ingeniería de software para sistemas embebidos, sistemas críticos y entrega continua de software”. XXI Workshop de Investigadores en Ciencias de la Computación (WICC 2019).
- Mascheroni, M.A; Irrazábal, E.; Carruthers, J.A.; & Pinto J.A. (octubre, 2019). “Rapid Releases and Testing Problems at the industry: A survey”. XXV Congreso Argentino de Ciencias de la Computación (CACIC 2019), pp. 787-796.
- Irrazábal, E.; Bernal, R.A.; Ríos, J.L.; Mascheroni, M.A.; Sambrana, I.; Alonso, J.M.; & Acevedo, J. (mayo, 2020). “Ingeniería de software para sistemas embebidos, sistemas críticos e Internet de las Cosas”. XXII Workshop de Investigadores en Ciencias de la Computación (WICC 2020), pp. 567-573.
- Mascheroni, M.A; Irrazábal, E.; & Rossi, G. (mayo, 2021). “Continuous Testing Improvement Model”. 2nd ACM/IEEE International Conference on Automation of Software Test (AST 2021).
- Mascheroni, M.A.; Orué Mascheroni, C.A.; Lezcano Airaldi, A.; & Irrazábal, E. (in press) “Problems and Solutions for Continuous Testing at the Industry: A Survey”. Revista Computación y Sistemas.
- Irrazábal, E.; Mascheroni, M.A. (in press) “Introducción a las Pruebas Continuas”. Editorial Eudene.

1.7. Organización del documento

El resto de los capítulos de esta tesis, se organizan de la siguiente manera:

- El capítulo 2 ofrece una visión detallada de los conceptos relacionados con el desarrollo continuo de software: Integración Continua, Despliegue Continuo, Entrega Continua y Pruebas Continuas.
- En el capítulo 3 se detallan los resultados de una revisión sistemática de la literatura y una encuesta, realizadas para identificar y relevar el problema de las pruebas en el desarrollo continuo de software, detallando las dificultades que existen en la actualidad.
- En el capítulo 4 se presenta el Modelo de Mejora para Pruebas Continuas propuesto.

- En el capítulo 5 se describe la herramienta desarrollada como soporte metodológico para el modelo propuesto.
- El capítulo 6 presenta la validación de este trabajo, mediante la aplicación del método de Investigación-Acción.
- Finalmente, el capítulo 7 perfila las conclusiones del trabajo, analizando la consecución de objetivos, el contraste de resultados y las líneas de trabajo futuras.

Adicionalmente, se presentan una serie de anexos que han sido separados de la disertación principal con objeto de mejorar la comprensión de la exposición y cuyo contenido se detalla a continuación:

- Anexo A: detalla las fases durante el desarrollo de la revisión sistemática de la literatura, los artículos seleccionados y la evaluación de la calidad de estos.
- Anexo B: presenta los resultados correspondientes a los proyectos encuestados y el cuestionario realizado a los mismos.
- Anexo C: profundiza el análisis realizado entre los problemas de Pruebas Continuas encontrados en la literatura académica y los detectados en la industria.

2. ESTADO DE LA CUESTIÓN

En este capítulo se presenta el marco teórico, introduciendo la calidad en el desarrollo de software, las metodologías ágiles y los procesos que forman parte del desarrollo continuo de software: Integración Continua, Despliegue Continuo, Pruebas Continuas y Entrega Continua.

2.1. La calidad en el desarrollo del software

De acuerdo con Parnas [46], en la Ingeniería de Software se trabaja mayormente con la "construcción de aplicaciones software multiversión". Por lo tanto, muchas de las actividades asociadas con una aplicación software provocan revisiones para mejorar la funcionalidad o para corregir errores.

La calidad es un concepto complejo y multidimensional [47]. Garvin [48] ha hecho un intento de ordenar las diferentes perspectivas de calidad:

- Perspectiva trascendental: donde la calidad es reconocida, pero no descrita. Se realizan definiciones subjetivas y no cuantificables.
- Perspectiva del usuario: la calidad está directamente relacionada con la satisfacción de las necesidades del usuario. Características como la fiabilidad, el rendimiento o la eficiencia son las tenidas en cuenta en esta perspectiva.
- Perspectiva de la fabricación o del proceso: se centra en la conformidad con las especificaciones y la capacidad para producir software de acuerdo con el proceso de desarrollo implementado en la empresa. Se tienen en cuenta características como la tasa de defectos o los costes de retrabajo.
- Perspectiva de producto: especifica que las características de calidad del producto se definen a partir de las características de sus partes. Por ejemplo, líneas de código, complejidad, diseño.
- Perspectiva basada en aspectos económicos: miden la calidad de acuerdo a costes, precios, productividad, etc.

En el caso del desarrollo software, la calidad puede estudiarse desde el punto de vista de la calidad del producto software y la calidad del proceso de desarrollo software [49].

2.1.1. ISO/IEC 9126 y 25000

La norma ISO/IEC 9126, ahora englobado en el proyecto SQuaRE para el desarrollo de la norma ISO/IEC 25000, establecía un modelo de calidad en el que se recogen las investigaciones de multitud de modelos de calidad propuestos por los investigadores durante los últimos 30 años para la caracterización de la calidad del producto software [50].

Este estándar propone un modelo de calidad dividido en tres aproximaciones: interna (calidad del código), externa (calidad en la ejecución) y en uso, que influyen entre sí (Fig. 6). Así, la calidad interna influye a la externa y ésta a la calidad en uso.

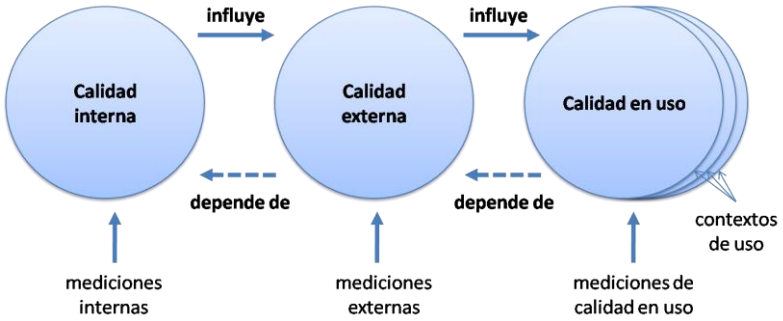


Fig. 6. Las tres perspectivas de calidad de ISO/IEC 25000.

La evaluación de la calidad de un producto software requiere de un modelo de calidad definido. Un modelo de calidad está compuesto de características, que se dividen en subcaracterísticas. Finalmente, las subcaracterísticas se componen de atributos, que obtienen sus valores de mediciones sobre el software. En este caso, el modelo establece diez características, seis de ellas comunes a las vistas interna y externa y las otras cuatro propias de la vista en uso. Estas serán presentadas en la sección 2.6.

La nueva versión de la norma ISO/IEC 9126 (ISO/IEC 25010) [51] forma parte de las normas ISO/IEC 25000 SQuaRE (Software Product Quality Requeriments and Evaluation). El objetivo de esta serie es guiar el desarrollo de productos software siguiendo una especificación y evaluación de requisitos de calidad. Así mismo, establece criterios para la especificación de requisitos de calidad de productos software, sus métricas y su evaluación.

La serie 25000 no establece los niveles de calidad deseables para cada proyecto, pero si recomienda que los requisitos de calidad deban ser proporcionales a las necesidades de la aplicación y lo crítico que sea el correcto funcionamiento del sistema implementado. Por lo tanto, será trabajo del jefe de proyecto determinar correctamente el nivel de calidad final que un producto software deberá alcanzar.

El principal objetivo de la norma ISO/IEC 25010 es proporcionar un marco para definir y evaluar la calidad del software. Para lograr esto, la norma define dos modelos. El primero es el modelo de la calidad del producto que proporciona un conjunto de características de calidad relacionadas con las propiedades estáticas y las propiedades dinámicas del software. Para ello, identifica ocho características que componen los modelos para evaluar la calidad del producto a diferencia de la norma ISO/IEC 9126 que definía seis): idoneidad funcional, eficiencia en el rendimiento, compatibilidad, usabilidad, confiabilidad, seguridad, mantenibilidad y portabilidad.

2.2. Metodologías ágiles

En febrero de 2001, tras una reunión celebrada en Utah (Estados Unidos), nace el término "ágil" aplicado al desarrollo de software [52]. En esta reunión, participaron un grupo de 17 expertos de la industria del software, incluyendo algunos de los creadores e impulsores de metodologías de software. Su objetivo fue esbozar los valores y principios que deberían permitir a los equipos desarrollar software rápidamente y respondiendo a los cambios que puedan surgir a lo largo del proyecto. Se pretendía ofrecer una alternativa a los procesos de desarrollo de software tradicionales, caracterizados por ser rígidos y dirigidos por la documentación que se genera en cada una de las actividades desarrolladas.

Tras esta reunión se creó la denominada Alianza Ágil[4], una organización sin fines de lucro dedicada a promover los conceptos relacionados con el desarrollo ágil de software, que ayuda a las organizaciones a adoptar dichos enfoques. El punto de partida fue el Manifiesto Ágil [52], un documento que resume la filosofía ágil.

Las metodologías ágiles traen consigo un conjunto de buenas prácticas y enfoques para mejorar los procesos de desarrollo [3]. Entre ellos se encuentran la Integración Continua de los componentes del software, la programación en pares, la documentación mediante historias de usuario, los equipos multidisciplinarios, entre otros.

2.2.1. El Manifiesto Ágil

Los valores definidos en el Manifiesto Ágil no se centran en prácticas, metodologías o procedimientos de trabajo, sino que persiguen un cambio de cultura organizativa basada en cuatro pilares [52].

[4] *Agile Alliance* - https://www.agilealliance.org/

Los pilares del Manifiesto Ágil son:

- **El individuo y las interacciones del equipo de desarrollo sobre el proceso y las herramientas**. Las personas son el principal factor de éxito en un proyecto software. Es más importante construir un buen equipo que construir el entorno. Muchas veces se comete el error de construir primero el entorno y esperar que el equipo se adapte automáticamente.
- **El software funcional con la documentación necesaria**. La regla a seguir es "no producir documentos a menos que sean necesarios de forma inmediata para tomar una decisión importante".
- **La colaboración con el cliente en lugar de la negociación basada en un contrato**. Se propone que exista una interacción constante entre el cliente y el equipo de desarrollo. Esta colaboración entre ambos será la que marque la marcha del proyecto y asegure su éxito.
- **La respuesta a los cambios en lugar de seguir estrictamente un plan**. La habilidad de responder a los cambios que puedan surgir a lo largo del proyecto determina también el éxito o fracaso del mismo. Por lo tanto, la planificación no debe ser estricta sino flexible y abierta.

2.2.2. Comparación con las metodologías tradicionales

La Tabla 2 recoge esquemáticamente las principales diferencias de las metodologías ágiles con respecto a las tradicionales ("no ágiles"). Estas diferencias afectan no sólo al proceso en sí, sino también al contexto del equipo y a su organización.

Tabla 2. Diferencias entre metodologías ágiles y no ágiles.

Metodologías Ágiles	Metodologías Tradicionales
Basadas en heurísticas provenientes de prácticas de producción de código	Basadas en normas provenientes de estándares seguidos por el entorno de desarrollo
Especialmente preparados para cambios durante el proyecto	Cierta resistencia a los cambios
Impuestas internamente (por el equipo)	Impuestas externamente

Metodologías Ágiles	Metodologías Tradicionales
Proceso menos controlado, con pocos principios	Proceso mucho más controlado, con numerosas políticas/normas
No existe un contrato tradicional, sino un contrato ágil no prefijado y flexible.	Existe un contrato prefijado
El cliente es parte del equipo de desarrollo	El cliente interactúa con el equipo de desarrollo mediante reuniones.
Grupos pequeños (< 10 integrantes)	Grupos grandes
Pocos roles	Más roles
Menos énfasis en la arquitectura del software	La arquitectura del software es esencial y se expresa mediante modelos

2.3. Integración Continua

La integración de los componentes de software está presente desde pequeños proyectos que pueden ser llevados a cabo por una sola persona, hasta una gran empresa con una arquitectura compleja donde trabajan cientos de desarrolladores. De este modo, la complejidad de la integración aumenta junto con la cantidad de personas trabajando en el proyecto y la cantidad de dependencias y otros sistemas relacionados. En procesos tradicionales, la integración se realiza al finalizar el proyecto, utilizando diferentes técnicas como *top-down, bottom-up, big-bang*, entre otras [53]. Sin embargo, esperar hasta el final del proyecto para realizar la integración puede generar problemas en términos de calidad, los cuales a su vez son costosos y producen demoras en los tiempos de entrega [54]. La Integración Continua (IC) es una de las prácticas utilizadas para resolver estos inconvenientes.

El término “Integración Continua” fue mencionado por primera vez en 1995, en el libro de Grady Booch [55]. En el mismo, Booch expresa que “el proceso macro del desarrollo orientado a objetos, consiste en un proceso continuo de integración”. Además, menciona que en intervalos regulares de tiempo, el proceso de Integración Continua produce nuevas versiones del software ejecutables que crecen en funcionalidad. Finalmente, Booch afirma que, gracias a estos hitos, se puede medir el progreso y la calidad atacando los riesgos de forma continua.

De acuerdo a Duvall et al. [9], Integración Continua es simplemente un avance en la evolución del proceso de integración del software. Al principio, la práctica de ejecutar

construcciones de código nocturnas automatizadas[5] ha sido considerada como una buena opción. Sin embargo, otras prácticas similares han sido propuestas en diferentes libros y artículos. En el libro “Microsoft Secrets” [56], se menciona la práctica de ejecutar construcciones diarias[6] en Microsoft. Asimismo, Steve McConnell [57] recomienda la práctica de ejecutar construcciones diarias y pruebas de humo como parte de los proyectos de desarrollo de software. Con la venida de las metodologías ágiles y con IC como una práctica recomendada, las personas comenzaron a notar que la construcción no debería ser diaria, sino “continua” [9].

Finalmente, Martin Fowler describe a IC como “una práctica del desarrollo de software, donde miembros de un equipo integran su trabajo con frecuencia. Usualmente, cada persona realiza al menos una integración en el día, lo que conduce a múltiples integraciones diarias. Cada integración es verificada por un proceso de construcción automatizado (que incluye pruebas), para detectar errores de integración tan pronto como sea posible” [4].

2.3.1. El proceso de Integración Continua y sus componentes

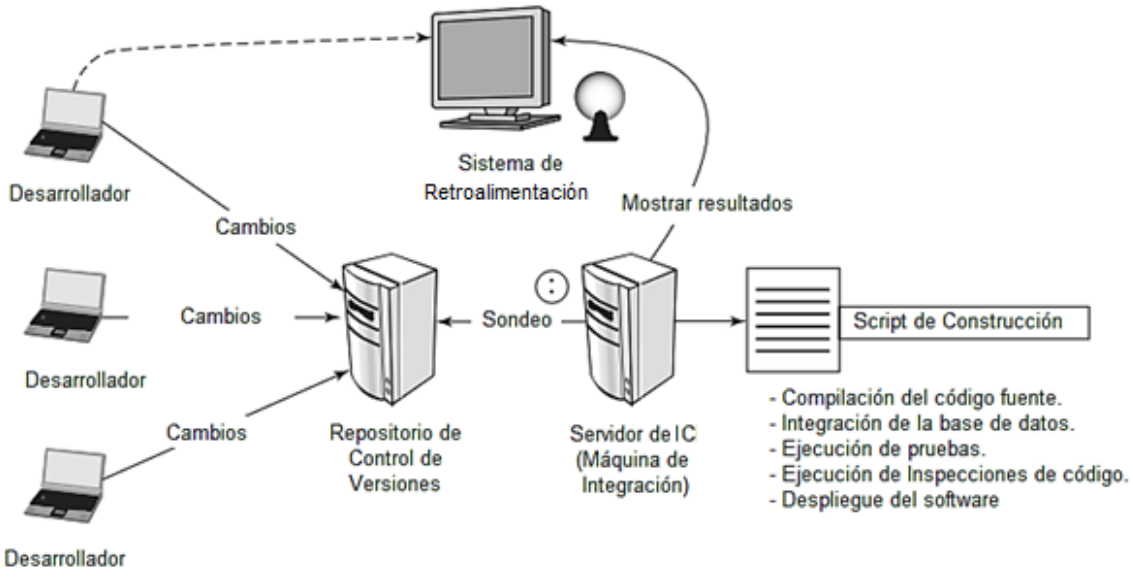


Fig. 7. Sistema de Integración Continua.

Un escenario de Integración Continua comienza cuando un desarrollador realiza una confirmación de cambios o *commit*[7] hacia el repositorio donde se encuentra el código fuente. Sin

[5] *Nightly Builds*
[6] *Daily Builds*
[7] El commit, es el proceso de confirmar un conjunto de cambios de forma permanente desde el entorno de desarrollo local del programador hacia un servidor central compartido por el equipo de trabajo.

embargo, esta confirmación no solo se limita a código fuente del programa principal; también abarca cambios en esquemas de bases de datos, archivos de configuración o documentación. La Fig. 7 muestra un escenario completo de Integración Continua y sus componentes. Los pasos en el proceso de Integración Continua siguen el siguiente flujo [9]:

1. Un desarrollador introduce un cambio en el repositorio de control de versiones. Mientras tanto, el servidor de IC en la máquina de integración se encuentra sondeando este repositorio para detectar nuevos cambios (por ejemplo, cada 15 segundos).
2. Cuando el servidor de Integración Continua detecta los cambios introducidos en el repositorio de control de versiones, el mismo los obtiene y ejecuta un script de construcción, el cual los integra al software.
3. El servidor de Integración Continua envía de un correo electrónico a los miembros del proyecto o personas involucradas comunicando el resultado.
4. El servidor de Integración Continua continúa realizando el sondeo para detectar nuevos cambios en el repositorio.

A continuación, se describen los actores y componentes involucrados en la Fig. 7.

- **Desarrollador**: la persona que modifica el código fuente y realiza la confirmación de cambios. Antes de enviar los cambios al repositorio de control de versiones debe ejecutar una construcción privada en su entorno local [58], para detectar cualquier problema de compilación o defecto antes de la integración. La construcción privada puede realizarse en cualquier momento sin afectar a los pasos siguientes del proceso de Integración Continua, ya que el mismo comienza una vez que los cambios son introducidos en el repositorio de control de versiones.
- **Repositorio de control de versiones**: un repositorio de control de versiones es simplemente un repositorio que tiene integrado un sistema de control de versiones (SCV).[8]
- **Servidor de IC**: ejecuta la integración cuando nuevos cambios son introducidos en el repositorio de control de versiones. Normalmente está configurado para detectar la introducción de estos cambios cada cierto periodo de tiempo. El servidor de Integración Continua obtiene el código fuente y ejecuta el script de construcción. Asimismo, estos servidores pueden ser configurados para ejecutar el proceso de IC con cierta frecuencia (como por ejemplo cada hora) en lugar de realizarlo con cada cambio introducido. Pero,

[8] Ejemplos de herramientas de control de versiones: Git, SubVersion.

según Duvall [9], eso no es Integración Continua. Existen una gran variedad de herramientas para operar como servidores de IC[9]. El servidor de Integración Continua debe ser una máquina distinta, cuya única responsabilidad es la de integrar software y realizar la construcción.

- **Script de construcción**: puede ser uno o varios scripts, que se utilizan para compilar, probar, inspeccionar y desplegar el software. Existen herramientas que sirven para crear un script de construcción[10], pero no proveen Integración Continua por sí mismas.
- **Sistema de retroalimentación**: Uno de los principales propósitos de la Integración Continua es comunicar los resultados de las integraciones, ya que esto logra identificar la introducción de errores con los últimos cambios introducidos. El sistema de retroalimentación permite enviar mensajes a los responsables de la construcción, con los resultados de la misma, tan pronto como finalice el proceso.

El proceso de Integración Continua, abarca una serie de etapas [4]: compilación del código fuente, integración de la base de datos, ejecución de las pruebas, ejecución de inspecciones y despliegue del software. Este proceso es puesto en marcha con cada confirmación de cambios y sus etapas son ejecutadas como parte del script de construcción. Paul Duvall [9], idealiza esto con el concepto de "botón de integración" (ver Fig. 8).

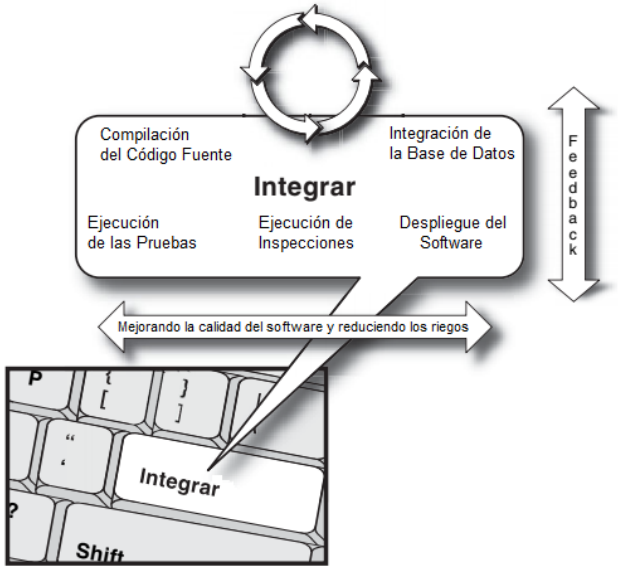


Fig. 8. Visualización del botón de integración [9].

[9] Ejemplos de herramientas para CI: Jenkins, Bamboo, CruiseControl, TravisCI
[10] Ejemplos de herramientas de construcción de código: Maven, Ant, NAnt, MSBuild y Rake.

La **compilación del código fuente** [59], es una de las etapas comunes en un proceso de Integración Continua e incluye la creación de código ejecutable a partir del código fuente para después pasar a la **integración de la base de datos**. Posteriormente, se ejecutan una serie de **pruebas unitarias**. El objetivo de las mismas es probar pequeños componentes del código, para asegurar que la integración se haya realizado con éxito. Al mismo tiempo o luego de la ejecución de estas pruebas, pueden ejecutarse **inspecciones**, las cuales permiten identificar problemas relacionados a la mantenibilidad del código fuente como, por ejemplo, su complejidad, acoplamiento, cohesión, o dependencias [60].

Una vez que los cambios introducidos pasaron por todas las etapas mencionadas, se realiza un **despliegue del sistema software** resultante, para que pueda ser probado por un equipo especializado en pruebas manuales o por más pruebas automatizadas de aceptación.

Todo este flujo puede verse en la Fig. 9. Mientras las primeras tres etapas deben ser tareas automáticas, el despliegue y la ejecución de pruebas siguientes pueden serlas como no. Sin embargo, para implementar Despliegue Continuo, el despliegue también debe ser automático.

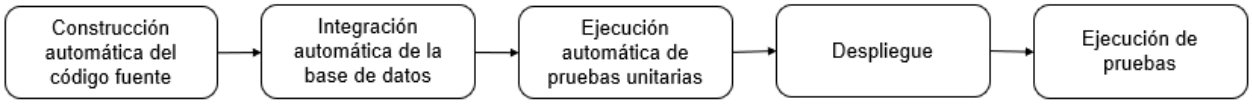


Fig. 9. Flujo de Integración Continua.

2.3.2. Retroalimentación continua

Según Duvall [9], “la retroalimentación es el corazón de Integración Continua”. La principal razón por la cual un equipo de trabajo implementa IC es porque desea detectar errores lo antes posible y esto solo se logra con retroalimentación rápida. Por ejemplo, si no se sabe el resultado de la ejecución de las pruebas de aceptación hasta luego de unas horas de la ejecución, no se pueden tomar acciones inmediatas y solucionar el problema antes que se propague.

El propósito de la retroalimentación es crear un sistema de notificación que genere una acción lo más rápido y preciso posible [15]. Además, la información tiene que ser la correcta, a las personas indicadas, en el momento exacto y utilizando el medio correcto. La Integración Continua es una herramienta que permite hacerlo, y, en consecuencia, la retroalimentación es continua. La Fig. 10 ilustra este proceso.

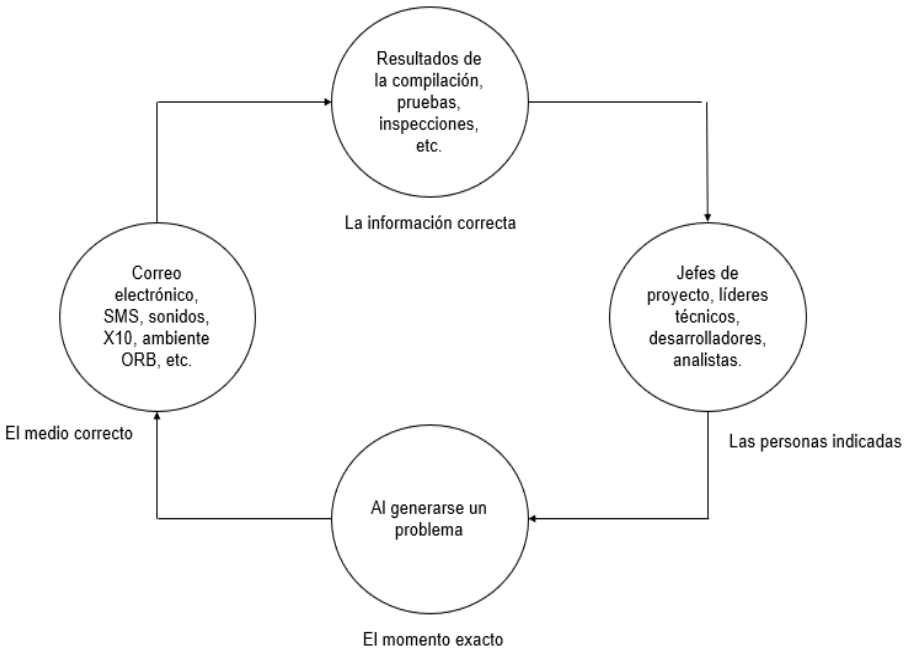


Fig. 10. Retroalimentación continua [9].

2.4. Despliegue Continuo

Se denomina despliegue del software al proceso de hacer que un producto software esté activo y funcionando en un entorno de trabajo, que podrá ser de pruebas o de producción. No solo es necesario integrar el software de manera frecuente, sino también desplegarlo a producción continuamente. Es decir, se necesita tanto de un proceso de Integración Continua como de uno de Despliegue Continuo (DC).

Paul Duvall y Steve Matyas [9] definen a Despliegue Continuo como “una culminación de prácticas y pasos que permiten lanzar software funcional en cualquier momento y lugar, con el menos esfuerzo posible”.

Los principios de Despliegue Continuo se encuentran en los procesos de desarrollo de software ágil y Lean[11], como se muestra en la Fig. 11.

La tendencia actual de las metodologías de desarrollo de software utilizadas en la industria es el desarrollo de software ágil [61]. De Cesare et al. y Williams concluyeron que, desde el punto de vista de la organización, el principio del desarrollo ágil más importante es el siguiente: “la principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software de valor” [62].

[11] Lean es una filosofía de trabajo que busca satisfacer las necesidades y expectativas del Cliente, con el menor consumo de recursos, a través de la continua eliminación de desperdicios, variaciones e inflexibilidades

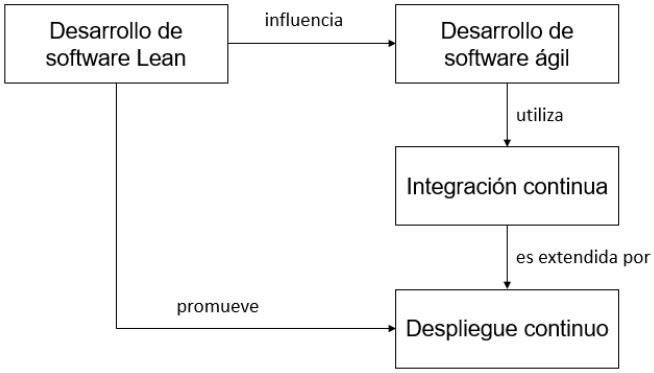


Fig. 11. Mapa conceptual del proceso de Despliegue Continuo [24].

Si bien los desarrolladores de software ágil buscan comprender cómo entregar software de trabajo con frecuencia, existe poca investigación sobre cómo utilizar metodologías y procesos ágiles para lograrlo [24]. El proceso de Despliegue Continuo fue la primera práctica propuesta que busca responder este interrogante, al permitir que los desarrolladores desplieguen el software con frecuencia y, por lo tanto, lo lacen a producción más rápidamente.

Por otro lado, el Desarrollo de Software Lean (DSL) es otro de los promotores de Despliegue Continuo. Se define como “la aplicación de los principios del sistema de desarrollo de productos de Toyota al desarrollo de software” [63]. Poppendieck y Poppendieck [63] proponen siete principios que forman la base del Desarrollo de Software Lean: eliminar el desperdicio, amplificar el aprendizaje, decidir lo más tarde posible, entregar lo más rápido posible, empoderar al equipo, construir integridad y ver el todo.Los principios del Desarrollo de Software Lean en conjunto con los métodos ágiles ayudan a los equipos de desarrollo a "entregar software lo más rápido posible" [24]. De acuerdo a Duvall et al. [9], Amazon, Google y eBay son algunas de las primeras organizaciones que comenzaron a lanzar software funcional a producción con frecuencia. De hecho, Tim O’Reilly reportó que Flickr se encontraba liberando versiones de su aplicación cada 30 minutos [67].

2.4.1. Despliegue tradicional y Despliegue Continuo

Lo que diferencia a Despliegue Continuo de los despliegues de software tradicionales principalmente, es la frecuencia de los despliegues [24]: en DC el software se despliega a producción con más frecuencia que en un despliegue tradicional. Las diferencias claves entre estos dos tipos de despliegue se resumen en la Tabla 3.

Tabla 3. Comparación entre despliegue de software tradicional y Despliegue Continuo.

Característica	Despliegue tradicional	Despliegue Continuo
Frecuencia de los despliegues	Puede ir de 1 a 6 meses [68], incluso más. La frecuencia ha ido aumentando, de meses (metodologías de cascada o proceso unificado) a semanas (metodologías ágiles) [69].	Diaria. Empresas como IMVU despliegan el software a producción hasta 50 veces al día usando Despliegue Continuo. Otro ejemplo es Facebook, que lo hace una vez al día [69].
Riesgo al realizar el despliegue	Alto. Dado que el despliegue es poco frecuente y que el software que se va a lanzar contiene muchos cambios, existe una mayor probabilidad de que el mismo contenga defectos [7].	Como los cambios son pequeños y los despliegues frecuentes, el riesgo es menor [24].
Feedback del cliente / usuarios finales	Se lo obtiene luego de largos periodos de tiempo, que dependen de la frecuencia de los despliegues [68].	Se lo obtiene muy rápidamente, ya que los clientes y usuarios perciben los cambios de manera constante [70], [71].
Funciones que no se terminan de desarrollar	Según el Standish Group [72], solo el 69% de las funciones o características que son solicitadas al inicio del proyecto, son desarrolladas.	Ayuda a disminuir el desarrollo de características innecesarias mediante la retroalimentación frecuente [63], [73].

2.5. Entrega Continua

La Entrega Continua (EC), es un enfoque de la Ingeniería de Software en el cual los equipos construyen el producto software funcional en ciclos muy cortos de tiempo, de tal manera que el mismo pueda ser lanzado a producción en cualquier momento [10], [13].Una de las características más importantes de la Entrega Continua es la confiabilidad que se tiene al lanzar una nueva funcionalidad o cambio en el software [74]. Por ese motivo, cada cambio introducido debe someterse a un proceso de pruebas. Así, cualquier introducción, modificación o borrado de una funcionalidad, es lanzada a producción cuando se lo desee buscando minimizar la cantidad de errores introducidos.

2.5.1. Entrega continua y Despliegue Continuo

Con Entrega Continua, el despliegue a producción es automático y se debe pasar previamente por etapas de construcción y diferentes tipos de pruebas. Es por ello que la Entrega Continua y el Despliegue Continuo son términos que usualmente son considerados iguales [74]. Sin embargo, Despliegue Continuo consiste en enviar a producción cualquier cambio que el desarrollador integre al repositorio principal. En cambio, Entrega Continua si bien permite hacerlo, la decisión final de ir a producción la tienen las partes interesadas como el cliente o dueños de producto. La

Tabla 4 muestra las principales diferencias y similitudes entre estos dos enfoques.

Tabla 4. Diferencias y similitudes entre Despliegue Continuo y Entrega Continua.

Despliegue Continuo	Entrega continua
Se enfoca en el despliegue a producción en el tiempo más rápido posible.	Se enfoca en la confiabilidad de introducir cambios a producción.
Cualquier cambio introducido se lanza como una nueva versión a producción.	El negocio decide el momento en el que una nueva funcionalidad sale a producción.
El proceso de despliegue a cualquier ambiente es automatizado.	El proceso de despliegue a cualquier ambiente es automatizado.
Ejecución de las pruebas más importantes para acelerar el tiempo del despliegue.	Ejecución de pruebas rigurosas para asegurar que el software no contiene errores.
No hay intervención humana en el flujo.	Se requiere la ejecución de un comando para hacer el despliegue a producción.

EC y DC son complementarios a IC, pero no se pueden hacer al mismo tiempo. Sin embargo, para ambos, el despliegue debe realizarse de manera automatizada. La diferencia es que, en Despliegue Continuo, el despliegue se ejecuta inmediatamente luego de la finalización de una etapa anterior (por ejemplo, la etapa de pruebas).

En cambio, en Entrega Continua, el despliegue es llevado a cabo cuando se presiona el botón de despliegue. Estas diferencias se ven en la Fig. 12.

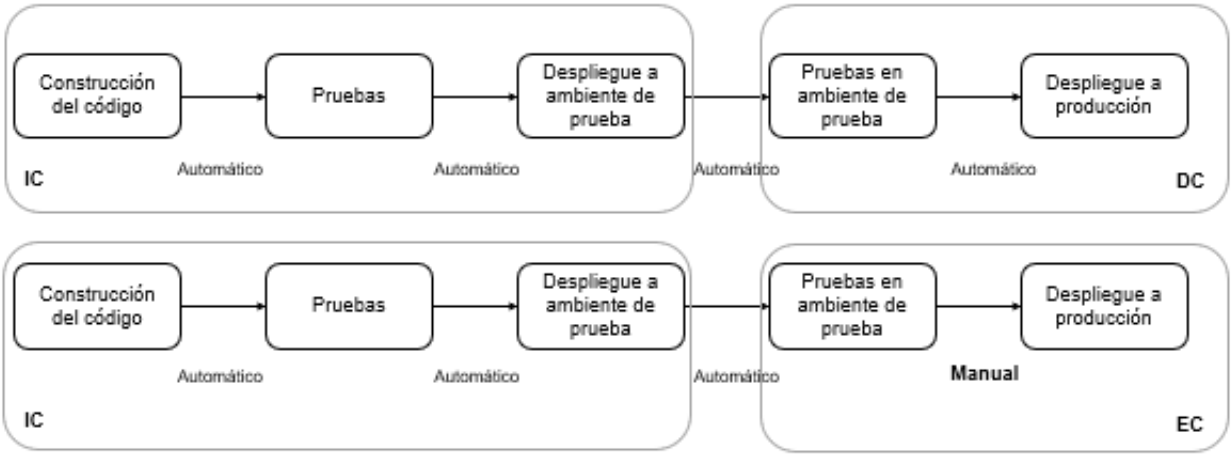


Fig. 12. Comparación en los flujos de Despliegue Continuo y Entrega Continua.

Si bien en la Fig. 12 la única diferencia apreciable es el despliegue a producción, las pruebas también son diferentes, ya que el Despliegue Continuo prioriza la velocidad, ejecutando solo las pruebas necesarias e involucrando al usuario final también como una etapa de pruebas más [75]. Por otro lado, en Entrega Continua se ejecutan la mayor cantidad de pruebas posibles buscando asegurar que el producto a lanzar a producción esté libre de errores [74].

2.5.2. Conducto de despliegue

El Conducto de Despliegue (DP) es la solución propuesta por Humble y Farley [7], como un enfoque holístico de extremo a extremo para la Entrega Continua.

El Conducto de Despliegue es un patrón de Entrega Continua para automatizar todas las etapas desde la integración de código en el repositorio de control de versiones, hasta que llegar manos de los usuarios finales [7]. Una manera gráfica de presentar el conducto de despliegue se ve en la Fig. 13.

Fig. 13. Anatomía del Conducto de Despliegue [7].

La entrada al DP se hace mediante la integración de código, donde el servidor de Integración Continua detecta los cambios introducidos en el repositorio de control de versiones. El código se construye y se ejecutan pruebas unitarias. Luego, se despliegan los cambios a un ambiente de pruebas y allí se ejecutan pruebas de aceptación. Posteriormente, los cambios se

vuelven a desplegar en un ambiente más similar a producción y se ejecutan pruebas exploratorias manuales. Finalmente, el software es lanzado a producción. Sin embargo, si se produjera un error en algunas de estas etapas, se notifica a los involucrados y se detiene todo el flujo. Todo este proceso se muestra en un diagrama de secuencia presentado en la Fig. 14.

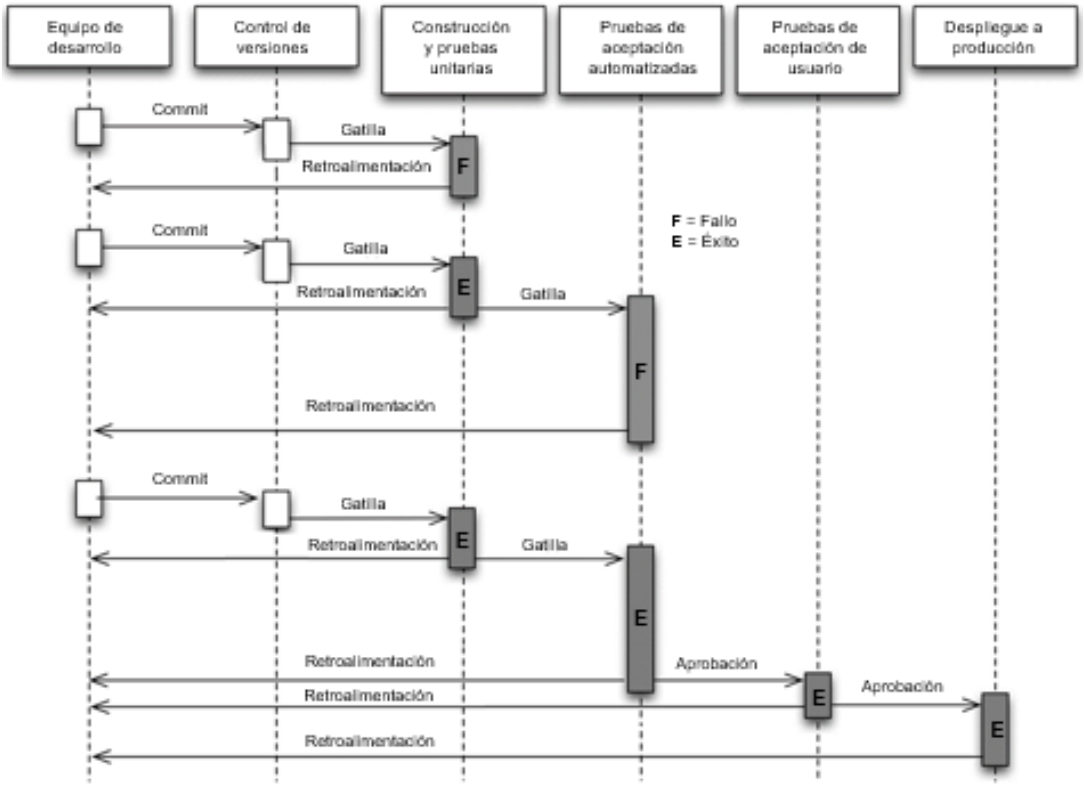


Fig. 14. Cambios en el código moviéndose a través del DP **[7]**.

Con este patrón se logra un entorno automatizado de compilación y pruebas de código dividido en etapas, que generan resultados rápida y eficientemente. Sin embargo, la implementación del conducto de despliegue depende de la necesidad de cada organización. Por ejemplo, Paddy Power[12], empresa en el rubro de apuestas online, ha implementado el DP como se muestra en la Fig. 15.

[12] Paddy Power. https://en.wikipedia.org/wiki/Paddy_Power

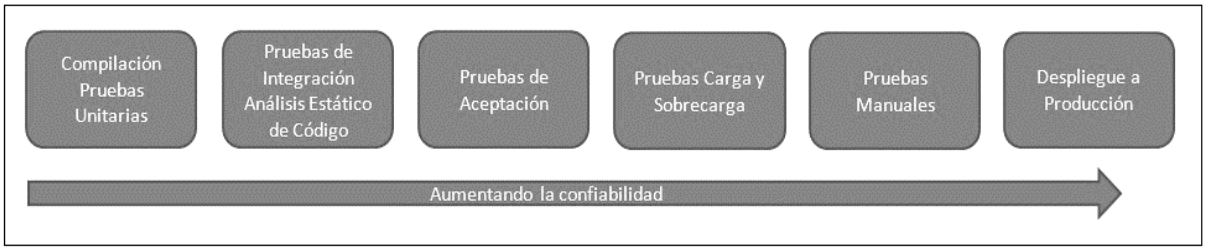


Fig. 15. Implementación del conducto de despliegue en Paddy Power [13].

La etapa que más varía en la implementación del conducto de despliegue es la de pruebas luego de los despliegues, variando en la cantidad de pruebas a las que se someten los cambios integrados. Los referentes del proceso de Entrega Continua, recomiendan las siguientes pruebas como necesarias en la implementación del conducto de despliegue [7]:

- Pruebas unitarias
- Inspecciones del código estático.
- Pruebas funcionales de interfaz gráfica de usuario (GUI) e interfaz de programación de aplicaciones (API).
- Pruebas no funcionales (capacidad, rendimiento, carga y sobrecarga).
- Pruebas exploratorias manuales.
- Pruebas de aceptación de usuario.

El éxito de adoptar prácticas continuas en las organizaciones depende mucho de la implementación del conducto de despliegue [76]. Por lo tanto, la elección de las herramientas e infraestructuras adecuadas para componer tal proceso también ayuda a mitigar algunos de los desafíos en la adopción e implementación de prácticas de IC, DC y EC [77]. En la Fig. 16 se muestran herramientas utilizadas en la implementación de DP, producto de una revisión sistemática de la literatura [77].

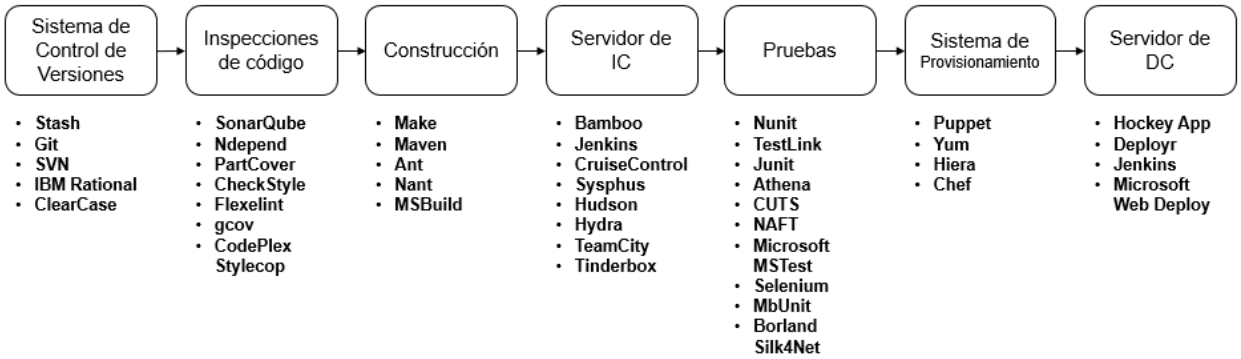


Fig. 16. Una visión general de las herramientas utilizadas para implementar DP.

2.6. Pruebas en entornos continuos

Las pruebas de software forman parte de una actividad realizada para evaluar la calidad de un producto y para mejorarlo mediante la identificación de defectos y problemas [28]. Para SWEBOK tienen tres características principales [28]:

- Dinámico: las pruebas se realizan solo sobre el programa en ejecución.
- Finito: las pruebas exhaustivas llevan mucho tiempo y es por ello que los ciclos de pruebas deben ser reducidos buscando cumplir un objetivo particular.
- Adecuado: existen muchas técnicas de pruebas y se deben elegir la más adecuadas para cada necesidad.
- Esperado: si no se cumple con una condición dada, los resultados indicaran un fallo o error.

Sin embargo, los mismos autores del SWEBOK, afirman que en la actualidad la visión de las pruebas de software ha evolucionado de tal manera que puede verse como una etapa complementaria [28]. Es decir que el proceso de pruebas inicia en las etapas de planificación de proyecto y concluye en las de mantenimiento. Asimismo, con la llegada de las metodologías ágiles, la verificación de software incluye tipos y niveles de pruebas que no solo se ejecutan con el software en ejecución, sino también durante su integración y construcción [9], [28], [78], [79].

Las pruebas son un componente clave en la medición de la calidad del software. Existen determinados tipos y niveles de pruebas, para diferentes atributos de calidad. La ISO/IEC 25000, también denominado SQuaRE (System and Software Quality Requirements and Evaluation), está formada por un conjunto de normas internacionales orientadas a la calidad del software, teniendo como objetivo la organización y gestión de los estándares que especifican los requerimientos de calidad del software y su evaluación [51], [80], [81]. Una de estas normas, la ISO/IEC 25010 que se menciona en la sección 2.1.1, define un modelo de calidad para los productos software [51].

Cada atributo del modelo de calidad de producto definido por la ISO/IEC 25010, será evaluado mediante un tipo específico de prueba de software. En la actualidad, existen muchos tipos y niveles de pruebas [82], [83], pero no todos pueden abordarse del mismo modo en los ambientes ágiles [78], [79]. Inclusive, las pruebas deben cumplir ciertos requerimientos para ser implementadas en los entornos continuos. Uno de estos requerimientos es la rapidez en sus tiempos de ejecución. Humble y Farley afirman que se deben automatizar todos los niveles de pruebas posibles [7] y Shahin et al. realizaron un estudio que demuestra que la automatización de pruebas es una pieza clave en Entrega Continua [77]. Sin embargo, las pruebas exploratorias

manuales son importantes en EC y no pueden ser automatizadas ya que requieren de la experiencia de un equipo especialista en pruebas manuales calificado [7], [13], [77], [78]. De este modo, los autores de estas prácticas proponen niveles y tipos de pruebas como necesarias, las cuales se describen a continuación.

2.6.1. Inspecciones de código

Las inspecciones de código, también conocidas como análisis estático del código fuente [84] son una de las primeras etapas de verificación de calidad en los entornos de desarrollo continuo [7], [9]. La misma es utilizada para medir el atributo de mantenibilidad, dentro del modelo definido por ISO/IEC 25010 [51]. Se ejecuta justo después de la integración del código fuente al repositorio, ya sea a una rama[13] o al tronco[14].

La inspección de código fuente es un tipo de análisis de software que se lleva a cabo sin que programa esté en ejecución y por esta razón se lo llama “estático” [85]. En enfoques tradicionales, la inspección de código no es considerada un tipo de prueba, ya que se ejecuta antes de que concluya la etapa de desarrollo.

El análisis estático de código fuente se realiza únicamente de manera automática por varios motivos [9]:

- Las inspecciones de código manuales son propensas a errores y son repetitivas.
- Las revisiones de código entre pares pueden ser subjetivas.
- Son más rápidas que las manuales.
- Las herramientas de inspecciones de código permiten personalizarse y generar métricas, guardando los resultados de las diferentes ejecuciones también de manera automática.

Dentro de la medición de la mantenibilidad, se encuentran métricas que son medidas por diferentes herramientas de análisis estático del código fuente. Estas métricas son [86]:

- Sentencias de código fuente no comentadas (NCSS).
- Complejidad del código (cognitiva, ciclomática, de Halstead) - CC, MVG.
- Código duplicado.
- Código no utilizado.
- Validación de estilos.

[13] *Branch*
[14] *Trunk*

- Dependencias Cíclicas (CDC).
- Acoplamiento aferente (Ca) y eferente (Ce).
- Número de módulos (NOM) y líneas de código (LOC).
- Líneas de comentarios (COM).
- Cobertura de pruebas unitarias.
- Número de pruebas unitarias sin ejecutarse.

Existen herramientas que integran la medición de todas las métricas de mantenibilidad, como por ejemplo SonarQube, Sonargraph o inFusion [87]. SonarQube es una plataforma para gestionar la calidad del código. Sus características principales son la comprobación de grandes conjuntos de reglas de codificación y diseño y la recopilación de métricas [87]. SonarQube calcula el gráfico de dependencia del sistema analizado y proporciona una matriz de dependencia a través de su vista Diseño. En la Fig. 17 se muestran ejemplos de resultados de análisis de SonarQube.

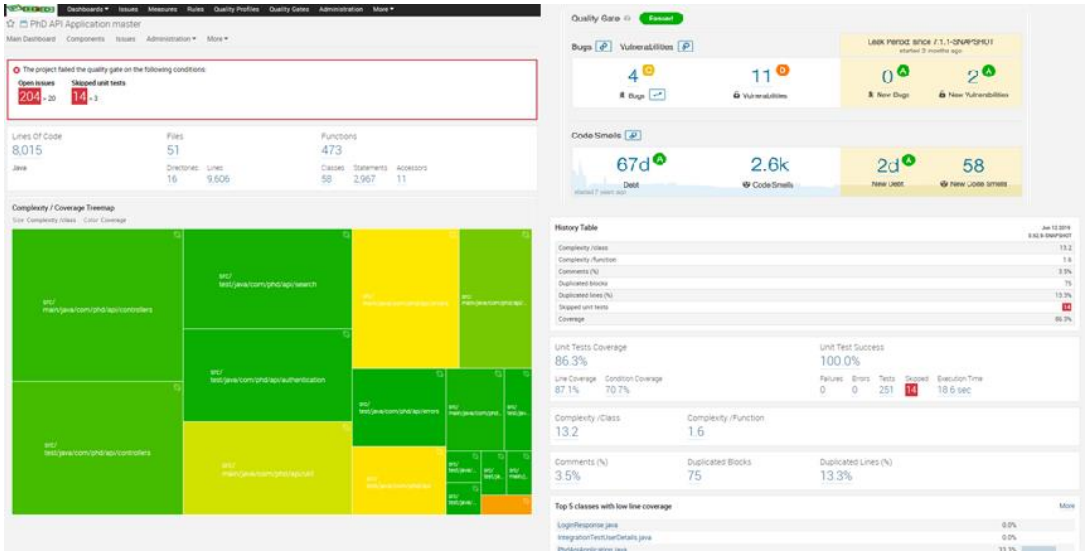


Fig. 17. Reporte de Inspección de código con SonarQube.

2.6.2. Pruebas unitarias

Las pruebas unitarias son verificaciones de pequeñas partes del código, por lo general funciones, métodos, subrutinas, o partes de ellas. Según SWEBOK, las pruebas unitarias verifican el funcionamiento de trozos de software de forma aislada y que se pueden probar por separado [28]. Dependiendo del contexto, estas piezas de código también podrían ser

subprogramas individuales o un componente más grande hecho de unidades estrechamente relacionadas. Es por ello que a veces a estas pruebas se las conoce como pruebas de componentes [82]. Sin embargo, otros autores utilizan el término pruebas de componentes para referirse a las pruebas de integración [9]. Por este motivo, en este trabajo no se utilizará el término *pruebas de componentes* para evitar ambigüedad.

Normalmente, las pruebas unitarias se realizan con el acceso al código que se está probando y con el soporte de herramientas de depuración, involucrando principalmente a los programadores que escribieron el código fuente.

Dentro de la norma 25010 [51], las pruebas unitarias se utilizan para medir el atributo de adecuación funcional. Es decir, que esta tarea no es solamente una verificación de una parte del software, sino que abarca un conjunto de actividades basadas en requisitos. Es así que el estándar IEEE 1008-87 define la etapa de pruebas unitarias de software, como un proceso que incluye etapas de planificación, desarrollo, ejecución y medición de resultados [88]. Este estándar, define un enfoque integrado que abarca la sistematización y documentación de pruebas unitarias. Describe un proceso de pruebas compuesto por una jerarquía de fases y actividades, definiendo un conjunto mínimo de tareas para cada actividad. Estas fases se presentan en la Fig. 18.

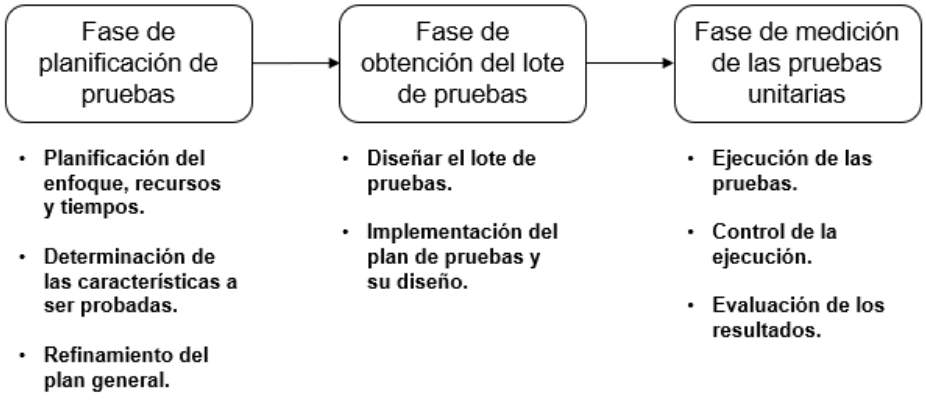


Fig. 18. Actividades del proceso de pruebas unitarias.

Sin embargo, para que las pruebas unitarias sean parte de un entorno continuo, las mismas deben automatizarse [9]. En la

Tabla 5 se listan las herramientas más conocidas para automatización de pruebas unitarias.

Tabla 5. Herramientas más conocidas para automatización de pruebas unitarias **[89]**.

Herramienta	Tecnología	Licencia
Atoum	PHP	Código abierto
C++ test	C/C++	Comercial
CppUnit	C/C++	LGPL
Cput	C/C++	Código abierto
Enhance PHP	PHP	Comercial
Go2xunit	Go	Código abierto
Jasmine	JavaScript	Código abierto
Jest	Java	Comercial
JMock	Java	Gratuito
Junit	Java	Código abierto
MbUnit	.NET	Gratuito
Mocha	JavaScript	Código abierto
Nunit	.NET	Código abierto
Parasoft C/C++ test	C/C++	Comercial
PHPUnit	PHP	Código abierto
Pytest	Python	MIT
PyUnit	Python	Código abierto
QA Systems Cantata	C/C++	Comercial
QUnit	JavaScript	Gratuito
SQLUnit	SQL	GLPv2
utMySQL	SQL	GPL
Xunit.net	.NET	Código abierto

A continuación, en el siguiente algoritmo (Algoritmo 1), se muestra una prueba unitaria escrita con la herramienta JUnit.

```
public class PHDTest {

  @Test
  public void verifyAdditionTest() {
    Calculator calculator = new Calculator();
    int numberOne = 8;
    int numberTwo = 4;
    int expectedResult = 12;
    int actualResult = calculator.add(numberOne, numberTwo);
    Assert.assertTrue(actualResult, expectedResult);
  }
}
```

Algoritmo 1. Ejemplo de prueba unitaria automatizada.

El caso de prueba presentado en el Algoritmo 1 verifica que un objeto de tipo Calculadora sume apropiadamente dos números. Sin embargo, se está realizando una verificación con un único conjunto de datos de entrada. Es por ello que para lograr la confiabilidad de las pruebas unitarias, deben utilizarse muchas combinaciones de datos de entrada [28]. Algunas pruebas unitarias requieren mínima dependencias de otras clases, pero no deben interactuar con entidades externas como sistemas de archivos o bases de datos, ya que en este caso serían pruebas de integración. Cuando una prueba unitaria requiere de datos, configuraciones o pasos previos para ejecutar las verificaciones, se utilizan objetos simulados[15] que son objetos simples simulando servicios reales, como una llamada a una base de datos.

El aspecto clave de una prueba unitaria es no depender de librerías externas las cuales tienden a incrementar la cantidad de tiempo que tarda la prueba en etapas de configuración y ejecución [9], [28], [88]. Es por ello, que esta fase debería poder ejecutarse rápida e inmediatamente después de la construcción del código fuente, con un tiempo máximo de ejecución de 5 minutos [7].

2.6.3. Pruebas de integración

Las pruebas de integración verifican interacciones entre diferentes módulos funcionales de una aplicación, o con sistemas externos como un sistema operativo, un sistema de archivos, la base de datos, entre otros [82]. Las estrategias de integración más modernas se basan en la arquitectura, lo que implica integrar los componentes o subsistemas de software basados en funcionalidades [28].

[15] *Mocks*

Al igual que las pruebas unitarias, para los entornos continuos las pruebas de integración también deben ser automatizadas [9]. De acuerdo a Jöngren [90], las pruebas de integración automatizadas resultan de la combinación del cumplimiento de la definición de pruebas de integración con la definición de pruebas unitarias automatizadas. Es decir, que es posible la utilización de herramientas de pruebas unitarias para automatizar las pruebas de integración. Otros autores como Humble y Farley [7], afirman que también pueden utilizarse herramientas de automatización de pruebas de aceptación para el mismo propósito. El Algoritmo 2 presenta un ejemplo de prueba de integración automatizada, para interactuar con una base de datos.

Cómo las pruebas de integración también verifican las interfaces entre módulos funcionales del mismo sistema, los cuales a su vez están compuestos por pequeñas unidades de código (funciones, métodos, subrutinas, etc.), las mismas deben ser ejecutadas luego de que las pruebas unitarias hayan finalizado [7], [9], [90].

```
public class DefaultWordDAOImplTest extends DatabaseTestCase {

  public DefaultWordDAOImplTest(String name) {
    super(name);
  }

  protected IDataSet getDataSet() throws Exception {
    return new FlatXmlDataSet(new File("test/conf/wseed.xml"));
  }

  protected IDatabaseConnection getConnection() throws Exception {
    final Class driverClass =
    Class.forName("org.gjt.mm.mysql.Driver");
    final Connection jdbcConnection =
    DriverManager.getConnection(
    "jdbc:mysql://localhost/words",
    "words", "words");
    return new DatabaseConnection(jdbcConnection);
  }

  public void testFindVerifyDefinition() throws Exception{
    final WordDAOImpl dao = new WordDAOImpl();
    final IWord wrd = dao.findWord("pugnacious");
    for(Iterator iter =
    wrd.getDefinitions().iterator();
    iter.hasNext();){
      IDefinition def = (IDefinition)iter.next();
      TestCase.assertEquals(
      "def is not Combative in nature; belligerent.",
      "Combative in nature; belligerent.", def.getDefinition());
    }
  }
}
```

Algoritmo 2. Ejemplo de prueba de integración automatizada.

2.6.4. Pruebas funcionales y no funcionales

En la literatura se encuentran diferentes definiciones para las pruebas de este grupo: pruebas de sistema, pruebas de aceptación, pruebas funcionales, pruebas de aceptación de usuario, pruebas no funcionales y pruebas de extremo a extremo. Esto quizás se deba a la evolución del proceso de pruebas.

Desde el punto de vista de las metodologías tradicionales, el Comité Internacional de Certificaciones de Pruebas de Software (ISTQB) define en el año 2002 a las pruebas de sistema como la verificación del comportamiento y funcionalidades de todo el sistema según lo definido en el alance del proyecto [82]. Esta etapa puede incluir pruebas basadas en riesgos, especificación de requisitos, procesos empresariales, casos de uso y otras descripciones de alto nivel del comportamiento del sistema.

Las pruebas de sistema deben incluir tanto verificaciones funcionales como no-funcionales, utilizando técnicas como caja negra y caja blanca[16]. Por otro lado, el ISTQB define a las pruebas de aceptación como una validación realizada por los usuarios finales o por el mismo cliente para corroborar que el sistema efectivamente cumple los criterios de aceptación y requerimientos planteados al inicio del proyecto [82].

Del mismo modo, en el año 2008 SWEBOK define a las pruebas de sistemas y a las pruebas de aceptación como lo hace el ISTQB [28]. En ambos, las pruebas de sistema conforman la última etapa de pruebas en el lado del equipo de desarrollo, ya que se verifica que el sistema a entregar cumple con las especificaciones, antes de que sea validado por el cliente o por el usuario final [28], [82]. En la mayoría de las organizaciones, esta etapa es realizada por un equipo especializado en pruebas, o por un servicio externo.

En el enfoque de la Entrega Continua, en el año 2007, Duvall et al. definen a las pruebas de aceptación y a las pruebas funcionales de forma análoga y añaden la idea de que estas pruebas también pueden ser llevadas a cabo dentro del equipo de desarrollo, desde un punto de vista de usuario final [9].

[16] Los términos caja negra y caja blanca son muy utilizados dentro de la teoría de sistemas tradicionales, en ingeniería de software, para referirse al tipo perspectiva con la cual es verificado un sistema. La verificación de una salida, en base a una entrada sin conocer los procedimientos que la transforman, se denomina caja negra. Cuando los procedimientos también se conocen, se denomina caja blanca.

En el año 2010 Humble y Farley utilizan el término “pruebas de aceptación” para referirse tanto a las pruebas funcionales como a las de sistema [7]. Esto se debe principalmente a dos razones:

1. En todas las definiciones anteriores, se realizan verificaciones de requerimientos funcionales y no-funcionales, variando únicamente la persona o grupo de personas que las llevan a cabo (desarrollador, especialista en pruebas, cliente, usuario final, etc.)
2. Si una característica del sistema no funciona como fue requerida (prueba funcional y sistema), entonces el sistema no es aceptado (prueba de aceptación). Es responsabilidad del equipo de desarrollo verificar que el sistema reúna los requisitos para ser aceptado por el cliente o por el usuario final.

Adicionalmente, en el año 2018, se menciona a las “pruebas de extremo a extremo” como parte de las pruebas de aceptación [91].

Finalmente, para diferenciar a estas pruebas de las realizadas por el usuario final, Humble y Farley llaman a estas últimas “pruebas de aceptación de usuario” [7].

Como este trabajo se centra en el desarrollo continuo de software, se utiliza el término **pruebas de aceptación** para referirse a la última etapa de pruebas realizada por el equipo de desarrollo, abarcando tanto a las **pruebas funcionales** como a las **pruebas no-funcionales**. Por otro lado, se utiliza el término **pruebas de aceptación de usuario** para referirse a las validaciones finales que realiza el cliente o usuario final.

2.6.4.1. Pruebas Funcionales

Es un tipo de prueba de aceptación, que al igual que las pruebas unitarias y de integración, se utilizan para medir el atributo de adecuación funcional, de la norma ISO 25010 [51]. La principal diferencia con estas dos últimas, es que se lleva a cabo con el sistema en funcionamiento, siendo una de las últimas etapas de verificación del lado del equipo de desarrollo [9]. Esto quiere decir que las pruebas funcionales son el mecanismo principal para verificar el cumplimiento de los requerimientos funcionales del sistema.

Dependiendo del sistema, las pruebas funcionales se realizan sobre una interfaz gráfica de usuario (GUI) o sobre una interfaz de programación de aplicaciones (API). La GUI puede ser un sitio web que se accede mediante un navegador, un programa de escritorio que se instala en el ordenador del usuario o una aplicación ejecutada desde un dispositivo móvil. La API puede ser un servicio web, una interfaz de líneas de comandos, o cualquier otro tipo de interfaz de sistema

que no sea gráfica. Sin embargo, los autores de los enfoques continuos solo mencionan y ejemplifican la automatización de sitios web mediante herramientas como Selenium[17] [7], [9]. Esto se debe a que los usuarios reales interactúan con la aplicación solamente a través de la GUI del sistema. Pero este enfoque no contempla a un sistema como usuario final de otro sistema, por definición de "actor" de acuerdo al estándar ISO/IEC 19501:2005 [92].

En el Algoritmo 3, se muestra un ejemplo de una prueba funcional automatizada, utilizando Selenium WebDriver. En el trabajo de Sabev y Grigorova [93], se presenta en profundidad las herramientas que existen en la actualidad para automatización de pruebas funcionales de GUI.

```
public class FunctionalTest {

  private WebDriver driver;

  @BeforeTest
  public void setUp() {
    driver = new FirefoxDriver();
  }

  @Test
  public void verifyLogin() {
    driver.get("http://localhost:8080/login.html");
    driver.findElement(By.id("username")).sendKeys("user1234");
    driver.findElement(By.id("password")).sendKeys("pass1234");
    driver.findElement(By.id("submit")).click();
    String expectedResult = "Welcome user1234";
    Assert.assertEquals(expectedResult, driver.getTitle());
  }

  @AfterTest
  public void tearDown() {
    driver.close();
  }
}
```

Algoritmo 3. Ejemplo de prueba funcional automatizada.

Las pruebas funcionales también pueden ser ejecutadas en etapas tempranas del ciclo de vida del software:

- Pruebas funcionales en el ambiente local del desarrollador: cuando el desarrollador desea probar su código, primero ejecuta la aplicación en su ambiente local con los nuevos cambios. Luego de verificar que se encuentra lista para ser probada, ejecuta los lotes de

[17] Selenium es un conjunto de herramientas con varias características que permite la automatización de acciones en un sitio web.

pruebas funcionales mediante la línea de comandos, o bien utilizando algún plugin desde el entorno de desarrollo integrado. Es importante que las pruebas funcionales sean configurables para apuntar al ambiente local (por ejemplo, localhost). La ventaja de este método es la facilidad de ejecución de las pruebas. La desventaja es que utiliza los recursos de la computadora donde el desarrollador se encuentra realizando sus tareas.

- Pruebas funcionales en el servidor de Integración Continua utilizando una rama funcional[18]: en los proyectos que se utilizan, el código puede ser enviado a la rama funcional dentro del repositorio sin necesidad de integrarlo a la rama principal. De este modo, es posible la ejecución de la aplicación desde la rama funcional en algún ambiente de desarrollo de bajo nivel y ejecutar las pruebas en el servidor de Integración Continua apuntando a esa versión. La principal ventaja de este método es que no requiere de recursos de ejecución en la computadora del desarrollador, pero su desventaja es que requiere un despliegue de la rama funcional al ambiente de desarrollo de bajo nivel.

2.6.4.2. Pruebas No funcionales

Es un tipo de prueba de aceptación que se utiliza para verificar requerimientos no funcionales y miden las características de la ISO 25010 que no son medidas por las pruebas descritas anteriormente. Estas características son: eficacia en el desempeño, usabilidad, seguridad, fiabilidad y portabilidad [51]. Adicionalmente, la mantenibilidad también es un requerimiento no funcional, pero medida mediante las inspecciones de código.

Los requerimientos no funcionales son muy importantes, ya que se han reportado casos de sistemas que habían fallado debido a que no podían soportar la carga, no eran seguros, eran muy lentos y hasta incluso no mantenibles debido a la baja calidad del código [7]. Sin embargo, por un lado, los autores de Integración Continua no mencionan a las pruebas no funcionales como un tipo de pruebas automatizables en el proceso de integración de código y por otro, los autores de Entrega Continua solo se centran en las pruebas de capacidad [7]. Según Chen [94], “si bien las pruebas unitarias y las pruebas de aceptación se han estudiado ampliamente en Entrega Continua, las prueba de requerimientos no funcionales han recibido una atención considerablemente menor”.

Con respecto a la eficacia en el desempeño de un sistema, se encuentran tres características: rendimiento, carga y capacidad [7]. El rendimiento es una medida del tiempo

[18] *Feature-branch* - https://martinfowler.com/bliki/FeatureBranch.html

transcurrido para procesar una transacción simple y se puede medir de forma aislada o con diferentes niveles de carga [95]. La carga, es el número de transacciones que un sistema puede procesar en un lapso de tiempo determinado y siempre está limitado por cuellos de botella en el sistema [95]. Finalmente, la máxima carga que un sistema puede soportar, para una carga de trabajo determinada, se denomina capacidad. Los clientes se centran principalmente en la capacidad del sistema [7] y para obtenerla se necesita medir las características mencionadas anteriormente, junto con otras relacionadas. La capacidad de un sistema abarca dos características principales [95]:

- Rendimiento: mide el tiempo transcurrido en el que un sistema procesa una transacción, en condiciones determinadas de trabajo.
- Transferencia de datos: mide el número de transacciones que un sistema puede procesar en un intervalo de tiempo.

La medición de la capacidad de un sistema incluye también determinar las características de la aplicación y para las verificaciones pueden utilizarse los siguientes tipos de pruebas [7]:

- Pruebas de escalabilidad: verifican cómo varía el tiempo de respuesta de un número N de peticiones al sistema, al agregar más servidores, servicios o hilos.
- Pruebas de longevidad: también llamada pruebas de picos, consiste en la ejecución del sistema durante un periodo largo de tiempo para verificar si el rendimiento varía en algún momento. El objetivo es detectar pérdidas de memoria[19] o problemas de estabilidad.
- Pruebas de transferencia de datos: se utiliza para detectar cuántas transacciones, mensajes, o peticiones a un sitio web, el sistema soporta en un intervalo de tiempo determinado.
- Pruebas de carga: verifica cómo varía el tiempo de respuesta del sistema cuando la carga en la aplicación aumenta. Dentro de este tipo de pruebas, se encuentran las pruebas de sobrecarga o estrés, que buscan hacer fallar todo el sistema mediante la utilización de una gran cantidad de carga. El objetivo de esta última es determinar la robustez de la aplicación en los momentos de carga extrema.

Existen un gran número de herramientas para realizar pruebas de capacidad. La Tabla 6 lista las herramientas más utilizadas del mercado. Por otro lado, la Fig. 19 muestra el resultado

[19] Una pérdida de memoria se produce cuando un programa no libera memoria, causando problemas de rendimiento y hasta una falla total en el sistema.

de la monitorización del tiempo de respuesta bajo diferentes parámetros de pruebas de capacidad, utilizando la herramienta Load Impact[20].

Tabla 6. Herramientas más conocidas para automatización de pruebas de capacidad [89], [96].

Herramienta	Tecnología	Licencia
Apache JMeter	Web	Código abierto
Google PageSpeed Insights	Web	Gratuito
GTMetrix	Web	Gratuito
Jetbrains doTrace	.NET	Comercial
LoadUI NG Pro	Web / API	Comercial
Microfocus LoadRunner	Web	Comercial
Open STA	Web / Windows desktop	Código abierto
Rational Performance Tester	Web / Server	Comercial
SmartMeter.io	Web	Comercial
WebLoad	Web / Mobile	Comercial
Gatling	Web	Código abierto
The Grinder	Web	Código abierto

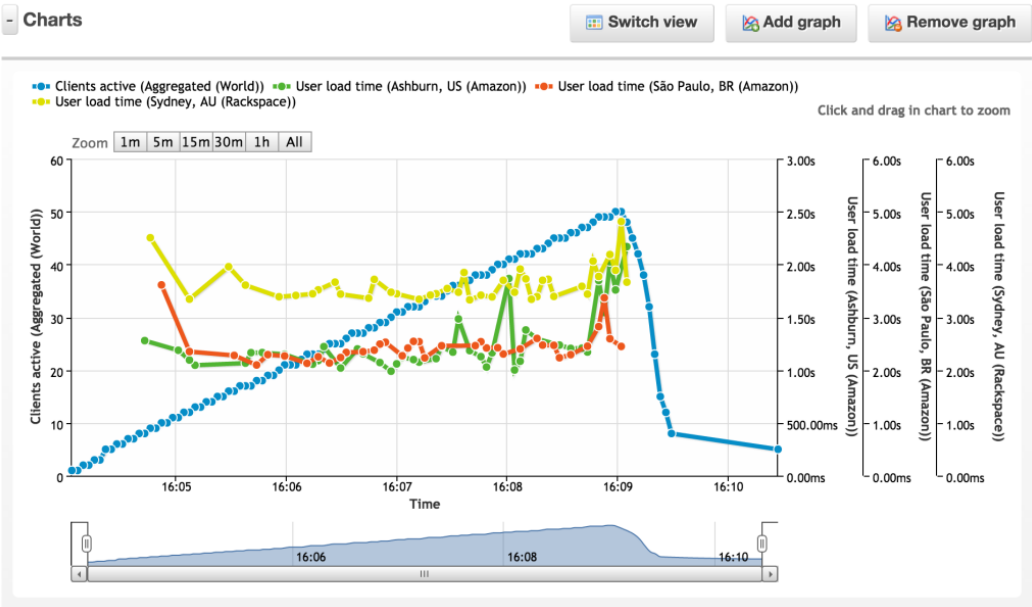


Fig. 19. Monitorización del tiempo de respuesta utilizando determinados escenarios de pruebas de capacidad.

[20] https://loadimpact.com/

Sin embargo, para incorporar las pruebas de capacidad en los entornos continuos, no basta con utilizar herramientas de automatización, sino que también deben convertirse en una etapa adicional del Conducto de Despliegue (ver Fig. 20) [7]. Esta etapa no solo contiene verificaciones de características de capacidad, como el rendimiento y las transacciones de datos, sino también la configuración del ambiente donde se realizarán las pruebas, el despliegue de los archivos ejecutables, pequeñas pruebas de humo[21] y finalmente la ejecución de las pruebas de capacidad.

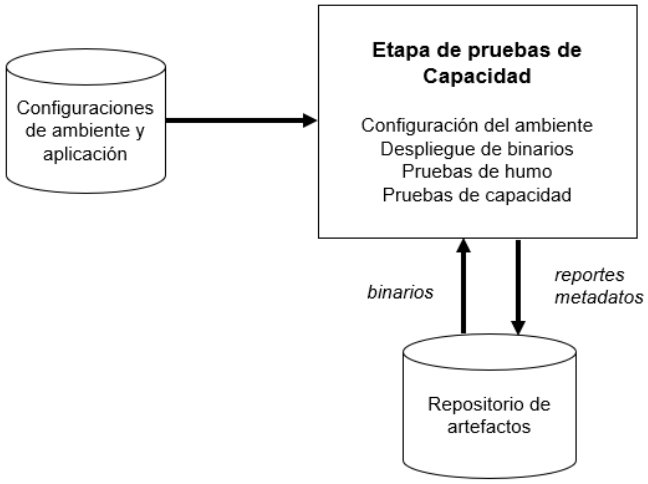


Fig. 20. Etapa de pruebas de capacidad en el Conducto de Despliegue [7].

2.6.5. Pruebas de aceptación del usuario

Existen tareas que no se pueden automatizar como la ejecución de pruebas basadas en la experiencia y la capacidad de análisis de un especialista en pruebas o las demostraciones de software funcional a los responsables del proyecto. En Entrega Continua, este grupo de tareas pertenecen a la etapa de pruebas de aceptación de usuario, que abarcan pruebas exploratorias, pruebas de usabilidad y demostraciones [7], [78], [79]. Esta etapa debe realizarse continuamente e incluirse en el DP [7].

En los entornos continuos, el rol del especialista en pruebas en este proceso no debería ser realizar una regresión total del sistema[22], sino validar el cumplimiento de los criterios de

[21] Las pruebas de humo (smoke testing), es una verificación rápida que se realiza para asegurarse que la funcionalidad básica de la aplicación se encuentre estable y responda al comportamiento esperado. Para ello, se utiliza un grupo seleccionado de pruebas, de el conjunto universal de pruebas funcionales.

[22] Las pruebas de regresión es una tarea que consiste en la ejecución total de todos los casos pruebas que verifican que se cumplan todos los requerimientos funcionales y no funcionales desde el inicio del proyecto.

aceptación de los requerimientos que han sido desarrollados e incluidos en la integración que se está realizando en ese momento, además de tareas que no pueden automatizarse como las pruebas exploratorias, las pruebas de usabilidad y las pruebas de aceptación de usuario [7].

Las pruebas exploratorias es un tipo de pruebas manuales en el cual el especialista en pruebas navega a través de la aplicación, utilizando su experiencia y capacidad de análisis para detectar nuevos casos de pruebas que pueden ser automatizados luego [97], [98]. Es un proceso creativo de aprendizaje, cuyo objetivo no es solo encontrar defectos, sino también aumentar la cobertura de las pruebas automatizadas mediante la detección de nuevos casos de pruebas [7], [98].

Las pruebas manuales realizadas por equipos de pruebas especializadas, por los dueños de producto o por el mismo cliente para validar que el software funcione de acuerdo a los requerimientos, se denominan pruebas de aceptación de usuario. Si el equipo de desarrollo se encuentra entregando software a producción siguiendo un contrato, las pruebas de aceptación de usuario son una etapa requerida para la aprobación final del software [79]. Es una manera de asegurar que los usuarios finales pueden realizar sus tareas [78].

Por otro lado, las pruebas de usabilidad deben incorporarse en los entornos continuos, para descubrir que tan fácil es para los usuarios finales cumplir con sus objetivos con el software que se está desarrollando [7]. Según la norma ISO 9241, la usabilidad se define como “el grado en el que un producto puede ser utilizado por usuarios específicos para conseguir objetivos específicos con efectividad, eficiencia y satisfacción en un determinado contexto de uso” [99]. Existen herramientas que permiten automatizar algunos aspectos de las pruebas de usabilidad [100], pero ninguna de ellas puede automatizar el proceso total. Las características de calidad que miden las pruebas de usabilidad son [100]–[102]: facilidad de aprender a usar el software, eficiencia del mismo, presentación visual de sus componentes de forma adecuada, manejo de los errores, satisfacción del usuario final, incluyendo el nivel de seguridad que ellos perciban de la aplicación.

Otra manera de realizar validaciones desde el punto de vista del usuario, del lado del negocio, es mediante las presentaciones del sistema a los clientes o dueños de producto. Estas presentaciones se conocen como demostraciones. Los equipos ágiles realizan este tipo de presentaciones al final de cada iteración para demostrar la nueva funcionalidad o características del sistema que serán entregadas [3]. La salida de esta reunión es la aceptación de la nueva funcionalidad, sugerencias para mejorarla, el rechazo de la misma o el surgimiento de nuevas ideas que serán planificadas y convertidas en nuevos desarrollos en el futuro. Humble y Farley,

consideran a las demostraciones como “el primer latido” de todo proyecto, al ser la primera vez que se tiene retroalimentación del cliente, usuario final o dueños de producto [7].

En general, la etapa de pruebas manuales verifica que el sistema funciona con los últimos cambios de código que representan los nuevos requerimientos funcionales y no funcionales que han sido desarrollados, detectando defectos que no hayan sido encontrados por las pruebas automatizadas y verificando que aporte valor para los usuarios finales [7].

2.7. Modelos de pruebas

La calidad de los productos software está fuertemente influenciada por la calidad del proceso que los generó [103]. En efecto, los procesos de pruebas contribuyen a la calidad del producto y representa un esfuerzo significativo en proyectos de desarrollo de software.

La mejora de procesos se ha vuelto cada vez más importante a lo largo de los años, con muchas organizaciones tratando de reducir sus costos de producción, mejorando la eficiencia de sus procesos de desarrollo [104]. Esta comprensión está marcada por una progresión de modelos de procesos de software, como ISO / IEC 12207, RUP, CMMI, MoProSoft o MPS.Br, entre otros. De todos ellos, el CMMI se considera el estándar de la industria más usado para la mejora de procesos de software [105].

Del mismo modo, diferentes modelos de procesos orientados a las pruebas se han desarrollado para gestionar la etapa de pruebas en las organizaciones que desarrollan software. García et al. [106] llevaron a cabo una revisión sistemática de la literatura que estudia los modelos de procesos de pruebas existentes en la actualidad y los resultados son los siguientes:

- TMM: Modelo de madurez de las pruebas. Basado en CMMI [107].
- TMMi: Modelo de madurez de las pruebas mejorado. Basado en CMM y TMM.
- TPI: Mejora de procesos de pruebas, que surge como una experiencia práctica.
- TIF: Un marco de trabajo para la mejora de procesos de las pruebas en software para sistemas embebidos.
- ATG-TPI: Un modelo de mejora de procesos para la generación de pruebas automatizadas. Se basa en TPI.
- BS 7925-2: Estándar para las pruebas de componentes de software.
- ISO/IEC 29119-2: Serie de estándares internacionales para las pruebas de software.
- ISTQB: Norma que certifica la calidad de los profesionales que intervienen en el proceso de pruebas de alto nivel.

- MB-V2M2: Modelo de madurez de verificación y validación basado en métricas, construido a partir de TMM.
- MMAST: Modelo de madurez para pruebas automatizadas de software.
- MPT.BR: Modelo de madurez de pruebas en Brasil, creado a partir de la fusión de la ISO/IEC 29119-2 y el TMM.
- STEP: Un proceso de evaluación y pruebas sistemáticas.
- TIM: Modelo de mejora de pruebas, que surge como una experiencia práctica.
- TAIM: Modelo de mejoras de pruebas automatizadas, basado en TMMi, TPI y TIM.
- TAP: Programa de evaluación de las pruebas.
- TCMM: Modelo de madurez y capacidad de las pruebas.
- TEST SPICE: Modelo de procesos de pruebas basado en la ISO/IEC 15504-5, TMMi y TAP.
- TOM: Modelo para la organización de las pruebas.
- TPI NEXT, proceso de mejoras para las pruebas orientadas por el negocio, basado en TPI.
- TPI-EMB: TPI para sistemas embebidos.
- MTPF: Marco de trabajo mínimo para las pruebas.
- TPI-Automotive: Un modelo basado en TPI para la generación automática de pruebas.

Teniendo en cuenta el grado de utilización de estos modelos por las organizaciones para sus procesos de desarrollo y por los autores para la construcción de otros modelos (Fig. 21), los más usados son TMM y TPI [106], [108].

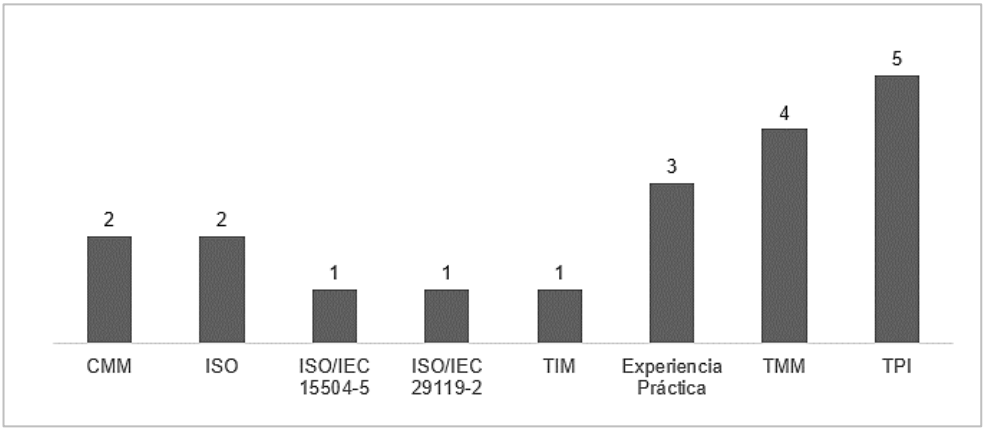


Fig. 21. Uso de modelos fuentes [106].

TMMi es un modelo desarrollado por el TMMi Foundation[23], evolución de TMM, que fue previamente creado por el Illinois Institute of Technology en 1996. Ambos modelos, basados en CMMI, definen cinco niveles de madurez específicos para las pruebas de software. Este modelo no solo se basa en CMMI, sino que también tiene el soporte del comité del ISTQB. Por otro lado, el TPI es un modelo de pruebas creado por la compañía española SOGETI[24], orientado a la capacidad y no a la madurez.

2.7.1. Modelo de madurez integrado para el proceso de pruebas (TMMi)

Las experiencias prácticas mencionadas en TMMi [105], demuestran que este modelo apoya a la implementación de un proceso de pruebas más efectivo y eficiente en la organización. El foco de las pruebas pasa de ser “la detección de defectos” a la “prevención de defectos”. El TMMi.

El TMMi ha sido desarrollado como un modelo por etapas. El modelo por etapas utiliza conjuntos predefinidos de áreas de proceso para definir una ruta de mejora para una organización. Este camino de mejora es descrito por un componente modelo llamado nivel de madurez. Un nivel de madurez es una meseta evolutiva bien definida para lograr procesos organizativos mejorados [107].

Dentro del alcance del TMMi, se encuentra [105]:

- **Ingeniería de software y sistemas**: El TMMi está destinado a apoyar las actividades de prueba y la mejora del proceso de pruebas tanto en la ingeniería de sistemas como en las disciplinas de ingeniería de software. La ingeniería de sistemas cubre el desarrollo de sistemas totales, que pueden incluir o no software. La ingeniería de software cubre el desarrollo de sistemas de software.
- **Niveles de pruebas**: Mientras que algunos modelos para la mejora del proceso de pruebas se centran principalmente en pruebas de alto nivel (como TPI) o abordan solo un aspecto de la prueba estructurada, TMMi abarca todos los niveles de pruebas (incluyendo las estáticas) y los aspectos de las pruebas estructuradas.
- **CMMI**: TMMi se posiciona como un modelo complementario a CMMI. En muchos casos, un nivel TMMi determinado necesita soporte específico de las áreas de proceso en su nivel CMMI correspondiente o de niveles CMMI más bajos. En casos

[23] http://www.tmmifoundation.org/
[24] https://www.sogeti.es/

excepcionales, incluso existe una relación con niveles más altos de CMMI. Las áreas y prácticas de proceso que se elaboran dentro del CMMI no se repiten en su mayoría en TMMi; solo están referenciados y reutilizadas.

- **Evaluaciones**: El TMMi proporciona un modelo de referencia para ser utilizado durante evaluaciones del proceso de pruebas cuyo objetivo es identificar oportunidades de mejora y comprender la posición de la organización con respecto al modelo. Como TMMi se puede utilizar junto con CMMI (en su versión por etapas), las evaluaciones de TMMi y CMMI usualmente se combinan, evaluando tanto el proceso de desarrollo como el proceso de pruebas. Como los modelos tienen una estructura similar y los vocabularios y objetivos del modelo se superponen, la capacitación paralela y las evaluaciones paralelas pueden ser llevabas a cabo por un equipo de evaluación. El TMMi también se puede utilizar para abordar problemas de prueba junto con modelos continuos. Las áreas de proceso superpuestas que se relacionan con las pruebas pueden evaluarse y mejorarse utilizando TMMi, mientras que otras áreas de proceso se incluyen en un modelo de alcance más amplio.
- **Enfoque de mejora**: TMMi proporciona un marco completo para ser utilizado como modelo de referencia durante la mejora del proceso de pruebas.

3. IMPORTANCIA DE LAS PRUEBAS EN LOS ENTORNOS DE DESARROLLO CONTINUO

En este capítulo, se presenta una introducción a la importancia de las pruebas en el desarrollo continuo de software. Luego, se describe el problema relacionado, detallando las dificultades que existen en la actualidad. Finalmente, se plantea la necesidad de un modelo de pruebas que resuelvan los inconvenientes mencionados.

3.1. Importancia de las pruebas en el desarrollo continuo de software

"Errar es humano, perdonar es divino" es una famosa frase del poeta Alexander Pope [109] que Jyothi, Krishna y Rao [110] la relacionan con las computadoras, en donde las mismas si bien parecen asombrosas en muchos sentidos, no son divinas y no perdonan los errores humanos.

Robert Martin, expresa lo siguiente [110]: "El software es una disciplina notablemente sensible. Si se modifica tan solo un bit del código fuente base, el software puede dejar de funcionar críticamente". Por otro lado, ISTQB afirma que todos el mundo en algún momento de su vida, experimenta una falla de software, desde un error en un dispositivo móvil hasta una transacción bancaria fallida [82]. "Un software que no funciona correctamente puede dar lugar a muchos problemas, incluyendo la pérdida de dinero y/o tiempo, daños personales o incluso la muerte" [82]. En este sentido, Kuhn, Wallace y Gallo [111], analizan 329 reportes de errores producidos en etapas de desarrollo e integración de un sistema distribuido de la NASA. Uno de estos errores, tiene que ver con la catástrofe del cohete Ariane 5 [112], que fue producida por un defecto de software: una conversión numérica errónea. Finalmente, en [113] se listan los defectos de software que tuvieron consecuencias significativas en diferentes áreas como la medicina, la milicia, las telecomunicaciones, el transporte, entre otras.

Todos estos ejemplos demuestran porque son tan necesarias las pruebas de software en cualquier proceso de desarrollo. En Integración Continua, se recomienda automatizar las pruebas unitarias, de integración y de rendimiento e incluirlas en los scripts de construcción [9]. Estos scripts luego tienen que ser ejecutados de manera local por los desarrolladores y posteriormente en el servidor de Integración Continua, al introducirse los cambios en el repositorio de control de versiones. En Entrega Continua, se busca automatizar más tipos y niveles de pruebas [7] e introducirlos en el Conducto de Despliegue como se muestra en la Fig. 13.

Las pruebas automatizadas, forman parte de uno de los componentes principales de Entrega Continua, ya que el objetivo del mismo, es realizar despliegues de productos fiables [7]. Sin embargo, la velocidad requerida para lanzar el producto al mercado con frecuencia en este tipo de enfoques, hace que las empresas pasen por alto muchos detalles a la hora de realizar pruebas [8]. Además, si bien existen un gran número de herramientas de automatización de pruebas, en la literatura pueden encontrarse revisiones sistemáticas que estudian los problemas y soluciones de Entrega Continua y mencionan que todavía se encuentran inconvenientes relacionados a las pruebas en este tipo de enfoque.

En la Tabla 7, se listan los problemas relacionados con la calidad de software, que han sido reportados en revisiones de la literatura, casos de estudio y artículos empíricos, cuyo objetivo era el estudio y la implementación de Entrega Continua.

Tabla 7. Problemas relacionados a la etapa de pruebas en Entrega Continua.

Problema	**Referencias**
Pruebas que consumen mucho tiempo de ejecución.	[27], [94], [114]
Pruebas no deterministas.	[27], [115], [116], [117], [118], [20]
Dificultades en pruebas automatizadas de interfaz gráfica de usuario.	[119], [120], [121], [122], [8], [123], [124], [125]
Resultados de ejecución de pruebas ambiguos.	[21], [116]
Dificultades en las pruebas de contenido dinámico web.	[122], [126]
Dificultades en las pruebas en Big Data.	[127]
Dificultades en las pruebas de datos.	[128]
Dificultades en la implementación de pruebas en dispositivos móviles.	[129]
Dificultades en pruebas automatizadas no funcionales.	[94], [117]

3.2. Revisión sistemática de la literatura en Pruebas Continuas

Como ha sido mencionado anteriormente, diferentes autores han reportados problemas en la implementación de Entrega Continua, especialmente en las etapas de integración y pruebas. Como parte de este trabajo, se ha llevado a cabo una revisión sistemática de la literatura (RSL), buscando propuestas, técnicas, nuevos enfoques, herramientas y otros tipos de soluciones para los diversos problemas mencionados. Además, el objetivo no ha sido solo identificar nuevas herramientas o técnicas, sino también comprender su relación con las Pruebas

Continuas (PC), para validar si las mismas son realmente el componente faltante de la Entrega Continua. También se ha investigado en los diferentes niveles de pruebas o etapas de pruebas que se utilizan en los entornos de desarrollo continuo de software. Finalmente, se ha profundizado en Pruebas Continuas, buscando sus diferentes componentes, sus limitaciones y si aún existen más problemas.

En la investigación basada en evidencias, el principal método utilizado es la revisión sistemática de la literatura (RSL). Una RSL proporciona una manera de identificar, evaluar e interpretar toda la investigación disponible sobre una cuestión de investigación, área o fenómeno de interés [33]. Las RSL son una revisión metodológicamente rigurosa de los resultados de investigación, cuyo objetivo final es servir a los investigadores para proporcionar soluciones de Ingeniería del Software apropiadas en un contexto específico.

Cabe resaltar, que antes de comenzar a realizar la revisión sistemática, es necesario comprobar que no se existe otra revisión de temática similar, para evitar la realización de un trabajo que ya ha sido realizado. Las RSL realizadas anteriormente sobre la temática se han centrado en temas de Entrega Continua tales como: características [6], [26], beneficios [6], [130], implementaciones técnicas [131], métodos de implementación [26], [132], problemas y causas en general [6], [12], [26] y soluciones en general [12]. Solo 3 de estos han cubierto en su investigación parte del proceso de pruebas en Entrega Continua. El primero de ellos [12], estudió los problemas y soluciones en su adopción y reportó algunos de los problemas de pruebas antes mencionados en la Tabla 7, pero que son parte de la etapa de adopción. El segundo artículo [26], estudió cómo los lanzamientos rápidos tienen repercusiones en la calidad del software, pero no analiza las posibles soluciones. El último de ellos [6], considera a las Pruebas Continuas como un factor clave en Entrega Continua. Sin embargo, solo describe desafíos y necesidades. Por lo tanto, cómo no existían RSL relacionadas a problemas y soluciones en Pruebas Continuas, se ha procedido a comenzar con el proceso de investigación.

La elaboración de RSL implica una serie de pasos bien estructurados y definidos [33]. A continuación, se describen estos pasos para la RSL realizada como parte de este trabajo.

3.2.1. Objetivos y cuestiones de investigación

El objetivo de la RSL es buscar soluciones que se hayan reportado para enfrentar los diferentes desafíos mencionados, como así también problemas sin solución. Asimismo, se intentan analizar los diferentes enfoques de Pruebas Continuas que podrían ser las soluciones

para esos desafíos. Del mismo modo, se investigan las diferentes etapas o niveles de pruebas que lo componen. Por lo tanto, se han propuesto las siguientes preguntas de investigación:

- **Cuestión 1**: ¿Existe una definición válida y aceptada para las Pruebas Continuas?
- **Cuestión 2**: ¿Qué tipos de pruebas o niveles de pruebas han sido implementadas en proyectos de desarrollo de software continuo?
- **Cuestión 3**: ¿Qué soluciones han sido reportadas para resolver los problemas relacionados a las pruebas en los proyectos de desarrollo continuo?
- **Cuestión 4**: ¿Existen todavía problemas de pruebas en proyectos de desarrollo continuo?

Con la cuestión 1, se pretende establecer qué son exactamente las Pruebas Continuas y si tienen una definición formal y aceptada tanto para estudios académicos como empíricos. En el mismo contexto, con la cuestión 2 se pretende establecer las etapas o niveles de prueba para Entrega Continua. El objetivo de la cuestión 3, es buscar cualquier tipo de soluciones (enfoques, herramientas, técnicas o mejores prácticas) que puedan utilizarse para afrontar los problemas de pruebas mencionados en la sección 3.1. Finalmente, el objetivo de la cuestión 4 es relevar una lista de problemas abiertos relacionado a las Pruebas Continuas.

Una vez establecidas las cuestiones de investigación, el siguiente paso es definir los criterios de búsqueda. En esta RSL se siguieron las fases sugeridas por Kitchenham [33]. La planificación, estrategia de búsqueda, la lista de artículos seleccionados y la evaluación de la calidad de los mismos se mencionan como parte del ANEXO A. El análisis y la discusión de los resultados obtenidos para cada cuestión de investigación se presentan a continuación.

3.2.2. Análisis y discusión de los resultados

En esta sección, se presentan y analizan los resultados de cada cuestión de investigación. Las referencias S1 a S55 corresponden a los artículos seleccionados presentados en el ANEXO A (Tabla 78).

3.2.2.1. Cuestión 1: ¿Existe una definición válida y aceptada para las Pruebas Continuas?

El término fue mencionado por primera vez por E. Smith en el año 2000 [133], como parte del proceso de Desarrollo Dirigido por Pruebas o TDD[25] considerando las pruebas unitarias.

[25] Siglas en inglés de Test Driven Development

Consistía en un proceso de pruebas que tenía que ser aplicado durante las etapas de desarrollo y de pruebas constantemente, como una regresión automática que se ejecutaba las 24 horas del día. Sin embargo, los resultados obtenidos de los trabajos seleccionados muestran que este concepto ha ido evolucionando durante los últimos años.

En el 2003, Saff y Ernst en S42 introdujeron el concepto de Pruebas Continuas como "un medio para reducir el tiempo perdido en la ejecución de pruebas unitarias". Utilizó la integración en tiempo real con el entorno de desarrollo para ejecutar asincrónicamente las pruebas que se aplican a la última versión del código, obteniendo eficiencia y seguridad al combinar las pruebas asincrónicas con las pruebas sincrónicas. Más tarde, entre los años 2004 y 2005 (S17), los mismos autores propusieron un complemento para el entorno de desarrollo integrado Eclipse, que usaba ciclos excesivos en la computadora de un desarrollador para ejecutar continuamente pruebas de regresión en segundo plano mientras el desarrollador escribe el código. Estas pruebas consistían en pruebas unitarias y pruebas de integración. Llamaron a este proceso como "Pruebas continuas". Proporcionaba retroalimentación rápida a los desarrolladores con respecto a los errores que introducían inadvertidamente mientras desarrollaban. Esta definición de Pruebas Continuas es la base de otras definiciones similares y también es utilizada por otros autores en la actualidad. Por ejemplo, en 2016, S25 usa el mismo término para referirse a "un proceso que proporciona retroalimentación rápida sobre la calidad del código al ejecutar automáticamente pruebas de regresión en segundo plano mientras el desarrollador está cambiando el código fuente".

En 2010, S29 toma la misma definición de Pruebas Continuas de S42 y S17: "ejecutar casos de prueba todo el tiempo durante el proceso de desarrollo para garantizar la calidad del sistema software que se construye". También presentaron el concepto de “Pruebas Continuas eternas”, incluyendo no solo la etapa de pruebas unitarias, sino también las siguientes: pruebas de especificación, pruebas de diseño, pruebas de codificación, pruebas de validación, pruebas funcionales, pruebas no funcionales, pruebas de despliegue, pruebas de operación, pruebas de soporte y pruebas de mantenimiento.

En 2011, utilizando el proceso de Pruebas Continuas presentado por Smith [133], S6 presenta “Pruebas Continuas para la computación en la nube”[26] probar aplicaciones SaaS. Como las aplicaciones pueden estar compuestas de servicios, las Pruebas Continuas pueden aplicarse antes y después de la aplicación y la composición del servicio e incluso durante su ejecución como un servicio de monitoreo. Más tarde, en 2013, para Google (S30), las Pruebas Continuas

[26] Más conocido en inglés como cloud computing

significan la "ejecución de cualquier tipo de prueba lo antes posible detectando cualquier tipo de problema relacionado con un cambio realizado por un desarrollador".

Entre 2015 y 2016, aparecieron nuevos enfoques de Pruebas Continuas: S10, S13, S14, S53 y S55. En S10 significa "usar la automatización para mejorar significativamente la velocidad de las pruebas al adoptar un enfoque de desplazamiento a la izquierda, el cuál integra las fases de control de calidad y desarrollo". Este enfoque puede incluir flujos de trabajo automatizados de pruebas que se pueden combinar con métricas para proporcionar una imagen clara de la calidad del software que se entrega. Según los autores de S10, utilizar un enfoque de Pruebas Continuas proporciona a los equipos retroalimentación sobre la calidad del software que están desarrollando. También les permite realizar pruebas antes y con una mayor cobertura al eliminar los cuellos de botella de pruebas, como el acceso a entornos de prueba compartidos y/o tener que esperar a que la GUI se estabilice. Según S10, "las Pruebas Continuas se basan en automatizar los procesos de implementación y pruebas tanto como sea posible y garantizar que cada componente de la aplicación se pueda probar tan pronto como se desarrolle".

Para S14, Pruebas Continuas implica ejecutar las pruebas de inmediato al integrar los cambios de código con el sistema y actualizar los casos de pruebas para poder ejecutar pruebas de regresión en cualquier momento y verificar que esos cambios no rompan las funcionalidades existentes. Para S53, Pruebas Continuas significa automatizar cada caso de prueba. Los procesos de prueba manual deben ser evaluados para las posibilidades de automatización. Los procesos de entrega de software deberían permitir ejecutar la totalidad de las pruebas en cada compilación de software sin ninguna intervención del usuario, avanzando así hacia el último objetivo de poder entregar una versión de software con calidad rápidamente. Todo este principio de Pruebas Continuas presentado en S53 no solo mueve el proceso de pruebas al comienzo del ciclo de vida de desarrollo de software, sino que también permite que las pruebas se realicen en un sistema similar a producción (complementado por Despliegue Continuo).

S13 presenta Pruebas Continuas para el desarrollo de aplicaciones móviles como "el proceso de ejecutar todas las pruebas de forma continua, tanto en todas las ramas principales del sistema de control de versiones". La más importante de estas pruebas son las pruebas de compilación, las pruebas de integración y las pruebas de rendimiento en un laboratorio de dispositivos móviles. Finalmente, en S55, los autores citaron una parte del libro de Integración Continua de Duvall et al. [9], donde afirman que "en un entorno de Integración Continua, las pruebas deberían ejecutarse continuamente". Este enfoque incluye pruebas unitarias, pruebas de integración, pruebas funcionales, pruebas no funcionales y pruebas de aceptación.

Los resultados muestran que el concepto de Pruebas Continuas ha evolucionado durante los años. Al principio, solo se aplicaba a la ejecución de pruebas unitarias de forma continua, especialmente en la computadora del desarrollador mientras desarrollaba el código en segundo plano. Ahora, no se aplica solo a las pruebas unitarias, sino también a cualquier tipo de caso de prueba que se puede automatizar. Esto puede indicar que la inclusión de diferentes etapas de pruebas durante los años en las definiciones de Pruebas Continuas está relacionada con la aparición de herramientas de automatización que permiten a los equipos automatizar diferentes tipos de casos de prueba (ver Fig. 22). Sin embargo, la mayoría de estos artículos basaron sus enfoques de Pruebas Continuas en la definición de Saff y Ernst (S42) y algunos de ellos utilizan la definición propuesta por Smith (2000). Por lo tanto, se puede concluir que Pruebas Continuas es el proceso de ejecutar cualquier tipo de caso de prueba automatizado lo más rápido posible para proporcionar retroalimentación rápida al desarrollador y detectar problemas críticos antes de pasar a producción.

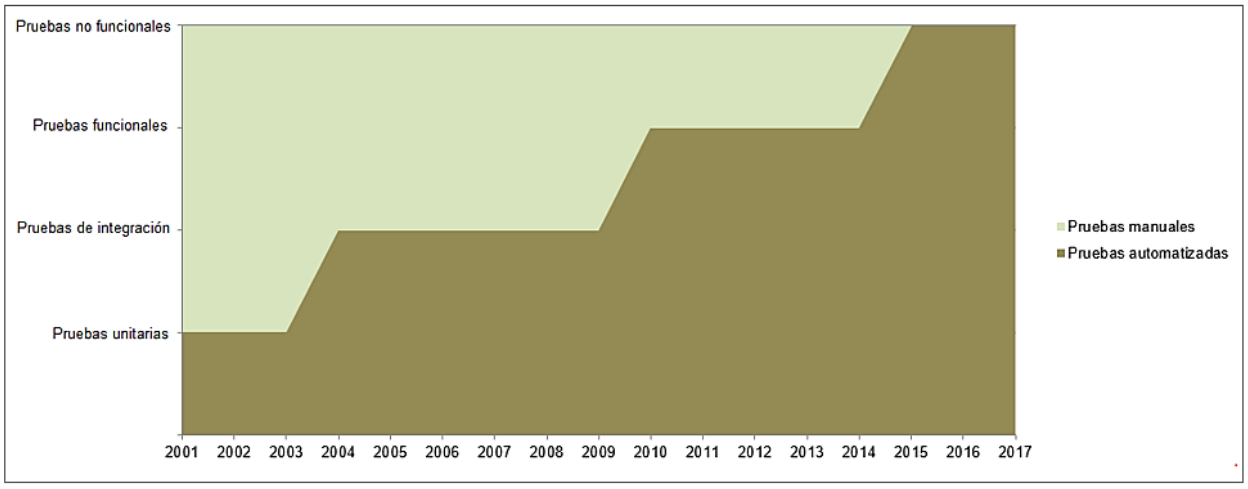


Fig. 22. Evolución de las pruebas automatizadas durante los años [11].

3.2.2.2. Cuestión 2: ¿Qué tipos de pruebas o niveles de pruebas han sido implementadas en proyectos de desarrollo de software continuo?

El ISTQB presenta 4 niveles de pruebas [82]: pruebas unitarias, prueba de integración, prueba de sistema y prueba de aceptación. Al mismo tiempo, estos niveles de prueba tienen subniveles de pruebas. Como se presenta en la sección 2.6, muchos de estos niveles se utilizan en Entrega Continua a través de la implementación de etapas específicas que incluyen diferentes tipos de pruebas.

Según los resultados obtenidos en la RSL, la primera etapa de Entrega Continua es la **revisión por pares**. Fue propuesto como una etapa de prueba (o etapa de aseguramiento de calidad) en EC por S2. Es una etapa manual, pero puede ser compatible con herramientas. Por ejemplo, S2 usa Gerrit como herramienta para la revisión por pares. Del mismo modo, S13 utiliza la **revisión de código** como un requisito antes de que cualquier cambio pueda enviarse a rama principal del sistema de control de versiones.

La segunda etapa de pruebas en Entrega Continua encontrada es **construcción** y **pruebas unitarias**. La mayoría de los estudios incluidos en esta RSL, que responden a la cuestión 2, han aplicado esta etapa de prueba (S2, S6, S9, S11, S12, S13, S14, S22, S32, S39 y S47). S22 ha extendido esta etapa mediante el uso de pruebas de mutación automatizadas. La prueba de mutación es un proceso mediante el cual el código existente se modifica de diferentes maneras específicas (por ejemplo, invirtiendo una prueba condicional de igual a no igual, o cambiando un valor verdadero a falso) y luego las pruebas unitarias se ejecutan nuevamente. Si el código modificado, o la mutación, no hace que una prueba falle, entonces la prueba funciona. Las herramientas relevadas para las pruebas unitarias son: JUnit (S2, S13), Robolectric (S13), NUnit (S9), Xcode (S13), Microsoft MSTest (S9), Android Studio (S13), XCTest (S13). Las herramientas utilizadas para las pruebas de mutación son: PIT Mutation Testing (S22), Ninja Turtles (S22) y Humbug (S22). S9, S11, S12, S22 y S47 también han agregado a este nivel, una etapa de **cobertura de código** que se ejecuta al mismo tiempo con las pruebas unitarias. La cobertura del código en pruebas unitarias se mide utilizando herramientas como JaCoCo (S22), CodePlex Stylecop (S9), Coverage.py (S22) o NCover (S22).

La tercera etapa de pruebas en Entrega Continua encontrada en la RSL es el **análisis estático del código fuente**. Fue implementado por S2, S12, S13 y S47. Es otra etapa automatizada que examina el código sin ejecutarlo, detectando problemas de estilos de codificación, alta complejidad, bloques de código duplicados, prácticas de codificación confusas, falta de documentación, etc. (ver sección 2.6.1). S22 establece que “el análisis estático permite que las revisiones manuales de código se concentren en diseños importantes y problemas de implementación, en lugar de aplicar estándares de codificación estilísticos”. Un nombre alternativo para esta etapa es Verificación de código (S47). La herramienta más implementada para esta etapa es SonarQube (S2, S12, S13 y S22).

La cuarta etapa reportada son las **pruebas de integración**. Es una etapa de pruebas automatizada implementada por S2, S6, S13, S14, S32 y S47. En esta etapa, los módulos de software individuales se combinan y se prueban como un grupo. JUnit (S2) y TestNG (S2) han sido reportados como herramientas para soportar esta etapa.

Algunos autores llaman a la quinta etapa como **pruebas de instalación** o **pruebas de despliegue**. Fue implementado por S14 y S22. El objetivo de esta etapa es verificar si la instalación del sistema o despliegue de software en un entorno específico se realizó correctamente. Es una etapa de verificación muy corta y las herramientas involucradas son las mismas que las que se utilizan para las pruebas unitarias o de integración.

La sexta etapa encontrada con la RSL la constituyen las **pruebas funcionales**. El objetivo de esta etapa es verificar que las funcionalidades del sistema funcionen como se espera. Esta etapa se implementó utilizando herramientas de prueba automatizadas por S2, S9, S11, S12, S13, S14, S22, S39, S43 y S47. Sin embargo, algunos de los estudios (S9 y S13) afirman que no es posible automatizar todos los casos de pruebas funcionales del proyecto. Por lo tanto, también utilizan pruebas manuales funcionales. Otros autores denominan a esta etapa como "pruebas de conformidad" (S13 y S43), "verificación de características" (S14), "pruebas funcionales de sistema" (S14) y "pruebas de aceptación funcionales" (S9, S11 y S22). S22 afirma que también es importante agregar *pruebas negativas* a esta etapa. Las pruebas negativas aseguran que el sistema pueda manejar entradas no válidas o comportamientos inesperados del usuario. Además, se han propuesto otras sub-etapas de pruebass como parte de la etapa de prueba funcionales, como las pruebas de instantáneas o snapshot testing (S13). El objetivo de estas últimas es generar capturas de pantalla de la aplicación, que luego se comparan, píxel por píxel, con versiones de instantáneas anteriores. Las herramientas utilizadas para la verificación en este nivel son: JUnit (S2), TestNG (S2), NUnit (S9), MbUnit (S9) o Borland Sil4Net (S9).

La séptima etapa son las **pruebas de seguridad**. Se utilizan para verificar que se cumplan los requisitos de seguridad como confidencialidad, integridad, autenticación, autorización, disponibilidad y no repudio. Esta etapa fue implementada por S9 y S39.

La octava etapa de pruebas encontrada es la constituida por **las pruebas de rendimiento, pruebas de carga y pruebas de estrés o sobrecarga** y fue implementada por S2, S9, S11 S13, S14, S39, S43 y S47. Esta etapa se implementa para determinar el comportamiento de un sistema en condiciones normales y anticipadas de carga máxima. Las herramientas utilizadas para esta etapa son Jmeter (S2) y Borland Silk Performer (S9). Una etapa adicional en este nivel son las **pruebas de capacidad**. Las pruebas de capacidad están dirigidas a verificar si la aplicación puede manejar la cantidad de tráfico que debe manejar. Estas últimas fueron implementadas por S9, S39 y S13. El conjunto de toda esta etapa se denomina **pruebas de CPLS** (siglas en inglés de capacidad, rendimiento, carga y esfuerzo) [11].

Finalmente, la última etapa de pruebas en Entrega Continua es la de **pruebas manuales exploratorias**. Las pruebas manuales se vuelven más importantes ya que las pruebas

automatizadas cubrirán los aspectos simples, dejando los problemas más alternativos o menos probables de ocurrir, sin descubrir. Esta etapa fue implementada por S9, S11 y S22.

Todas estas etapas de pruebas se han implementado para diferentes tipos de plataformas: aplicaciones web, aplicaciones móviles, computación en la nube, servicios web, aplicaciones de Big Data, etc. Las etapas de pruebas implementadas en los estudios se muestran en la Fig. 23, siendo las pruebas unitarias, las funcionales y las de CPLS las más utilizadas en entornos de desarrollo continuo de software.

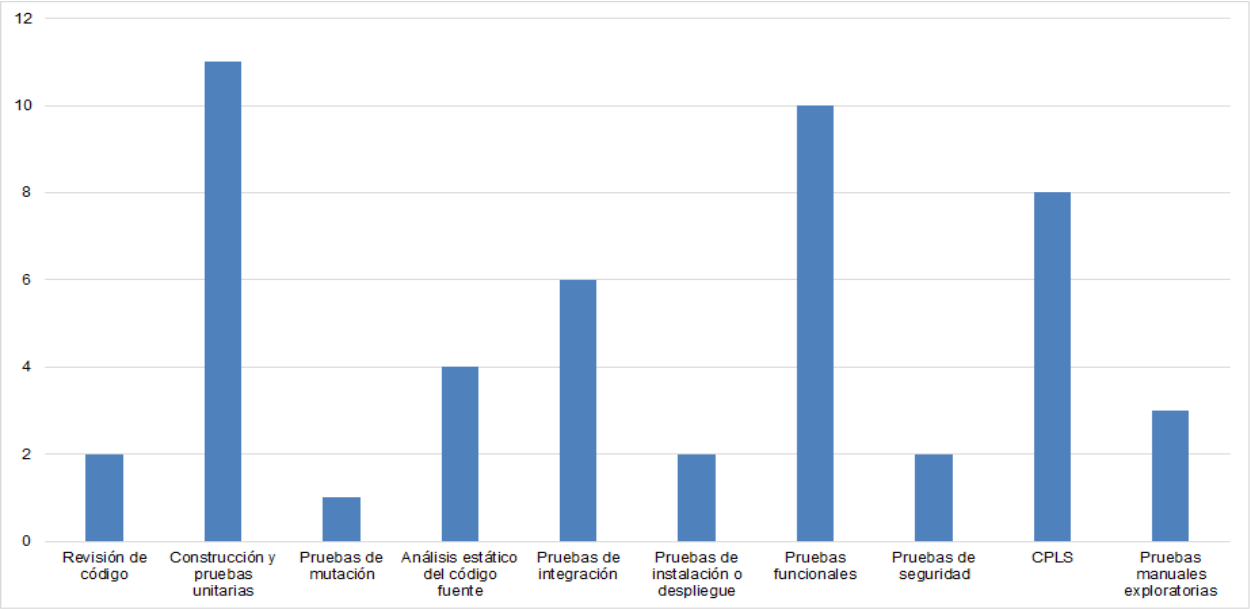


Fig. 23. Etapas de pruebas en entornos continuos implementas por los estudios de la RSL.

3.2.2.3. Cuestión 3: ¿Qué soluciones han sido reportadas para resolver los problemas relacionados a las pruebas en los proyectos de desarrollo continuo?

Diferentes tipos de soluciones han sido propuestas para resolver los problemas relacionados a las pruebas en los proyectos de desarrollo continuo, presentados en la Tabla 7. A continuación, se analizan esas soluciones para cada problema de forma separada.

PRUEBAS QUE CONSUMEN MUCHO TIEMPO DE EJECUCIÓN

En cualquier entorno de desarrollo continuo de software, los cambios se introducen con mayor frecuencia en el repositorio, por lo que es necesario ejecutar regresiones lo más rápido posible. Sin embargo, la ejecución de un lote de casos de prueba grande lleva demasiado tiempo,

incluso si los casos de prueba están automatizados. Además, este problema no está vinculado a un solo nivel de prueba, sino a todas las diferentes etapas de pruebas mencionadas. Por lo tanto, se analizan las diferentes soluciones agrupándolas por nivel de prueba

Pruebas unitarias

S1, S6 y S15 han propuesto el uso de técnicas de generación automática de casos de prueba para enfrentar este problema. Al tener un sistema completamente automatizado de generación de casos de prueba unitarios, es posible reducir considerablemente el costo de las pruebas de software y también facilita la escritura de los casos. Además, S6 presenta una técnica de priorización de pruebas unitarias, donde los casos de prueba se pueden clasificar para ayudar a los usuarios a seleccionar los casos de prueba más importantes para ejecutar primero y con frecuencia. Sin embargo, el desarrollador debe hacer la priorización manualmente y esa es una tarea que consume mucho tiempo.

S16 y S17 proponen un mecanismo que consiste en ejecutar pruebas unitarias en segundo plano mientras el desarrollador está codificando. S17 presenta un complemento de eclipse para proyectos Java que compila automáticamente el código cuando el usuario guarda el código en el búfer de la memoria y luego indica errores de compilación en el editor de texto de eclipse y en la lista de tareas, proporcionando una interfaz integrada para ejecutar lotes de pruebas JUnit. Del mismo modo, S16 utiliza el mismo enfoque para código .Net pero combinándolo con TDD, donde el desarrollador tiene que escribir las pruebas antes de escribir el código. Finalmente, S25 mejora este enfoque al agregar una estrategia orientada a la selección de pruebas. Cada módulo tiene su propio complemento y no es necesario ejecutar todas las pruebas unitarias, sino solo las relacionadas con el módulo afectado. Mediante el uso de convenciones de nomenclatura, el marco de trabajo propuesto puede encontrar el complemento de prueba para un módulo específico y las herramientas de cobertura de código pueden detectar clases modificadas. Por lo tanto, este enfoque impacta en los tiempos de ejecución de las pruebas unitarias, reduciéndolos y no impacta en la etapa de escritura del caso de prueba.

En la

Tabla 8 se muestra un resumen de las soluciones propuestas para las pruebas unitarias que requieren mucho tiempo de ejecución.

Tabla 8. Soluciones para pruebas unitarias que requieren mucho tiempo de ejecución.

Solución	Artículos que implementaron la solución	Grado en el cual la solución resuelve el problema
Generación de casos de prueba.	S1, S15	Parcial
Priorización de casos de prueba.	S6	Parcial
Ejecución de pruebas unitarias en segundo plano mientras se desarrolla el código.	S16, S17	Parcial
Ejecución de grupos de pruebas unitarias seleccionadas, en segundo plano mientras se desarrolla el código.	S25	Total

Pruebas funcionales

S6, S27 y S41 proponen técnicas de segmentación y agrupación de casos de prueba para enfrentar los tiempos de ejecución de prueba funcionales elevados. Las pruebas se agrupan en diferentes lotes según la funcionalidad y la velocidad. De esta manera, las pruebas más críticas se pueden ejecutar primero y los desarrolladores obtienen retroalimentación de ellas inmediatamente. Las pruebas no críticas y más lentas se ejecutan más tarde y solo si las primeras han pasado. Por lo tanto, la segmentación de prueba resuelve parcialmente el problema de pruebas funcionales que consumen mucho tiempo.

Otra alternativa para este problema es el uso de la paralelización (S6, S13, S22, S27, S41, S43, S50 y S56). La ejecución de pruebas automatizadas en paralelo (en lugar de en serie), disminuye la cantidad de tiempo al ejecutar las pruebas. La ejecución de las pruebas se distribuye a través de diferentes computadoras. S56 propone el uso de la virtualización como una alternativa para tener una sola computadora y distribuir la ejecución de la prueba a través de máquinas virtuales. Sin embargo, ambos enfoques requieren un hardware considerable.

S25 presenta un marco de trabajo llamado Morpheus que reduce el tiempo de retroalimentación y proporciona resultados de las pruebas solo para los cambios introducidos en el repositorio por el desarrollador. Reduce la cantidad de pruebas a ejecutar seleccionando solo

las que están relacionadas con el código fuente modificado. Una de las estrategias que han implementado para este marco es la “estrategia de selección de pruebas orientada a los requisitos”: el código fuente modificado se puede vincular con la historia del usuario, el requisito o la función de corrección de errores, que también se puede vincular con los casos de prueba. S44 propone otra técnica de selección de casos de prueba, donde la selección se basa en "el análisis de correlaciones entre fallas de casos de prueba y cambios en el código fuente". Sin embargo, ninguno de los dos enfoques ha resuelto el problema de los cambios que tienen un efecto en todo el sistema o en toda la aplicación que se está desarrollando.

S23, S24, S34, S45, S46 y S48 proponen la técnica de priorización automática de casos de prueba utilizando diferentes métodos de priorización. S23, S24 S45 y S48 implementaron la priorización basada en datos históricos para determinar un orden óptimo de pruebas de regresión en las ejecuciones de pruebas posteriores. Por otro lado, S34 utiliza la priorización basada en la perspectiva empresarial, la perspectiva de rendimiento y la perspectiva técnica. Finalmente, S46 presenta una herramienta llamada Rocket que ejecuta las pruebas que toman menos tiempo antes. Sin embargo, los resultados de la implementación de las técnicas de priorización muestran que, si bien resuelven la detección temprana de defectos críticos, no resuelven el problema que lleva mucho tiempo al ejecutar todo el conjunto de casos de prueba.

Se implementaron otras alternativas en S26, S27, S29, S52 y S56. S29 propone un enfoque llamado “Pruebas Continuas de larga duración” que utiliza métodos de inteligencia artificial (IA) en diferentes niveles. El enfoque consiste en ejecutar casos de prueba todo el tiempo en un servidor de compilación y detectar problemas mediante técnicas de IA. S52 propone una tecnología de optimización de lotes de pruebas llamada TITAN. S27 propone la rotación del navegador como una alternativa para aumentar la velocidad en la ejecución de pruebas de GUI web. Rotar los navegadores entre ejecuciones consecutivas puede lograr gradualmente la misma cobertura que ejecutar las pruebas en cada navegador para cada ejecución. S27 también propone el uso de API REST en las pruebas: algunos de los casos de prueba requieren la inicialización de la configuración mediante el uso de la interfaz de usuario en una página de configuración de la aplicación. La utilización de las API REST puede reducir la duración de la ejecución de algunos scripts de pruebas porque las operaciones son considerablemente más rápidas en comparación con la realización de ellas a través de la GUI.

S26 y S56 proponen pruebas como servicio (TaaS). Tanto S26 como S56 afirman que ejecutar pruebas automatizadas en paralelo es una de las mejores soluciones para pruebas que requieren mucho tiempo, pero esto requiere recursos de hardware. Una de las principales ventajas de TaaS sobre las pruebas tradicionales es su modelo escalable a través de la nube:

utiliza potencia informática, espacio en disco y memoria según los requisitos actuales, pero tiene la capacidad de aumentar la demanda muy rápidamente. Por lo tanto, ejecutar pruebas paralelas no es un problema. Además, TaaS admite múltiples tipos de pruebas automatizadas y reduce el costo de las pruebas internas.

Además, se proponen combinaciones de las técnicas mencionadas anteriormente. S31 presenta un enfoque que utiliza técnicas de selección y priorización de pruebas e integra diferentes métodos de aprendizaje automático. Estos métodos son: cobertura de prueba de código modificado, similitud textual entre pruebas y cambios, historial reciente de fallas o fallas de prueba y antigüedad de la prueba. Sin embargo, el enfoque conlleva los problemas de selección de prueba y priorización de prueba. Finalmente, el S30 muestra cómo Google enfrentó estos problemas al agregar la selección de pruebas, la priorización de pruebas y la ejecución de casos de prueba en la nube (TaaS).

En la

Tabla 9 se muestra un resumen de las soluciones propuestas para las pruebas funcionales que requieren mucho tiempo.

Tabla 9. Soluciones para pruebas funcionales que requieren mucho tiempo de ejecución.

Solución	**Artículos que implementaron la solución**	**Grado en el cual la solución resuelve el problema**
Agrupamiento o selección de casos de pruebas.	S6, S27, S41	Parcial
Paralelización de pruebas.	S6, S13, S22, S27, S41, S43, S50, S56	Total
Selección automática de casos de pruebas.	S25, S44	Parcial
Priorización automática de casos de pruebas.	S23, 24, S34, S45, S46, S48, S52	Parcial
Selección y priorización automática de casos de pruebas.	S31	Parcial
Ejecución de pruebas continuamente en un servidor.	S29	Parcial
Pruebas como un servicio (TaaS).	S26, S36	Total

Solución	Artículos que implementaron la solución	Grado en el cual la solución resuelve el problema
Selección y priorización automática de casos de pruebas, con ejecución de los mismos en paralelo utilizando TaaS.	S30	Total
Optimización de casos de pruebas.	S52	Parcial
Rotación de navegadores.	S27	Parcial
Uso de API REST.	S27	Parcial

Pruebas manuales

Las pruebas automatizadas han sido la solución para las pruebas manuales que tardan mucho tiempo durante años. Hay muchas herramientas que permiten a los desarrolladores automatizar casos de prueba tanto funcionales como no funcionales. Sin embargo, aparte de las pruebas automatizadas funcionales y no funcionales, se encontraron dos nuevos enfoques en la literatura:

1. Automatización de escenarios negativos (S22): las pruebas manuales se utilizan para realizar pruebas exploratorias que incluyen escenarios negativos, es decir, los que no forman parte del llamado "camino feliz"[27]. Sin embargo, las pruebas negativas pueden automatizarse y eso reduce el tiempo en la etapa de pruebas manuales.
2. Técnicas de priorización para las pruebas manuales del sistema de caja negra (S40): pruebas basadas en la cobertura, pruebas basadas en la diversidad y pruebas basadas en el riesgo. Los resultados muestran que el enfoque basado en el riesgo es más efectivo que los demás, en el contexto de entornos continuos de desarrollo de software. El enfoque basado en el riesgo utiliza la información histórica de detección de fallas y requiere acceso al resultado de ejecución anterior de los casos de prueba.

PRUEBAS NO DETERMINISTAS

Una característica importante de una prueba automatizada es su determinismo. Esto significa que una prueba siempre debe tener el mismo resultado cuando el código probado no

[27] *Happy Path*

cambia. Una prueba que falla al azar no es confiable y comúnmente se llama prueba no determinista[28]. Las pruebas automatizadas de este tipo ralentizan el progreso, no son confiables, esconden errores reales y cuestan costo debido a su mantenimiento.

S25 propone un marco de trabajo que ejecuta solo pruebas relacionadas con una nueva característica o una funcionalidad modificada. De esta manera, el número de pruebas no deterministas se reduce significativamente. Sin embargo, esta propuesta solo reduce la cantidad de pruebas no deterministas pero no las omite ni las repara. Del mismo modo, S31 propone un enfoque que realiza la selección de pruebas y la priorización de pruebas utilizando diferentes técnicas: cobertura de código modificado, similitud textual entre pruebas y cambios, historial reciente de fallas o fallas de prueba y antigüedad de la prueba. El enfoque rastrea las fallas por cobertura, texto e historia. Cuando encuentra fallas que no están relacionadas con el código nuevo o modificado en términos de cobertura o similitud de texto, significa que las pruebas son no deterministas. S45 también utiliza la priorización de pruebas y las técnicas de selección de pruebas, evitando errores en las compilaciones y retrasando la rápida respuesta que hace que la Integración Continua sea deseable.

Las pruebas automatizadas también pueden probarse para detectar si son deterministas o no (S41). Por ejemplo, S7 propone un mecanismo para la clasificación y análisis de pruebas no deterministas. Los autores de S7 han estudiado las causas comunes de este tipo de pruebas y sus soluciones. El objetivo de los autores es identificar enfoques que puedan revelar un comportamiento inestable y describir estrategias utilizadas por los desarrolladores para corregir las pruebas inestables. Sin embargo, las pruebas de pruebas introducen esfuerzo y tiempo.

Otra estrategia común es volver a ejecutar las pruebas. Google, por ejemplo (S30) tiene un sistema que recopila todas las pruebas que fallan durante el día y luego las vuelve a ejecutar por la noche. También usan un sistema de notificación donde se enumera el historial de ejecución de la prueba que ha fallado para que el desarrollador vea si es una prueba inestable. Google también ha implementado otras estrategias de mitigación para el control de la instabilidad de las pruebas. Si el nivel de inestabilidad de una prueba es demasiado alto el sistema de monitoreo pone en cuarentena la prueba y luego presenta un error para los desarrolladores.

Finalmente, S21 propone y evalúa enfoques para determinar si una falla o error al ejecutar una prueba se debe a un falso positivo o no:

[28] Más conocida en inglés como *flaky test*

- Posponer la re-ejecución de la prueba hasta el final de la ejecución del lote completo de pruebas. En este momento, hay más información disponible y las repeticiones se pueden evitar por completo.
- Volver a ejecutar las pruebas en un entorno diferente
- Revisar la cobertura de las pruebas con el último cambio: si una prueba que ha pasado en una revisión anterior falla en una nueva y si la ejecución de la prueba no depende de los cambios introducidos en esa revisión, definitivamente se puede concluir que la prueba no determinista. Si la prueba depende del cambio, no se puede concluir si la prueba es no determinista.

Existen diferentes soluciones para pruebas automatizadas no deterministas. Sin embargo, de acuerdo con los artículos analizados, todas las soluciones tienen ventajas y desventajas (ver Tabla 10) y todavía no existe una solución perfecta y aceptada para esta problemática.

Tabla 10. Ventajas y desventajas de las soluciones para las pruebas no deterministas.

Solución	**Ventajas**	**Desventajas**
Priorización y selección de pruebas.	1. Reduce el número de pruebas no deterministas en la ejecución del lote de pruebas.	1. Las pruebas no deterministas siguen existiendo. 2. Las pruebas no deterministas no son identificadas.
Ejecución de las pruebas solo para verificar la funcionalidad afectada por el código modificado.	1. Las pruebas no deterministas son fáciles de identificar e ignorar.	1. Las pruebas no deterministas siguen existiendo.
Probar las pruebas.	1. Las pruebas no deterministas pueden ser identificadas e ignoradas. 2. Es posible determinar la causa de la instabilidad de las pruebas. 3. Las pruebas no deterministas pueden ser eliminadas o solucionadas.	1. Costo y tiempo.

Solución	Ventajas	Desventajas
Volver a ejecutar las pruebas si fallan.	1. Reduce la cantidad de fallos producidos por pruebas no deterministas.	1. Tiempo. 2. Las pruebas no deterministas siguen existiendo.
Volver a ejecutar las pruebas si fallan, pero solo al final de la ejecución de todo el lote.	1. Reduce la cantidad de fallos producidos por pruebas no deterministas. 2. Es posible determinar la causa de la instabilidad de las pruebas.	1. Tiempo. 2. Las pruebas no deterministas siguen existiendo.

DIFICULTADES EN PRUEBAS AUTOMATIZADAS DE INTERFAZ GRÁFICA DE USUARIO

De acuerdo a lo indicado por Alégroth, Feldt y Ryrholm: “Las pruebas de alto nivel, como las pruebas de aceptación de interfaz gráfica de usuario, se realizan principalmente con prácticas manuales que a menudo son costosas, tediosas y propensas a errores” [119]. La automatización de pruebas se ha propuesto como una alternativa para resolver estos problemas. Sin embargo, la interfaz de usuario es la parte de una aplicación que cambia con mayor frecuencia y puede conducir a pruebas automatizadas que producen falsos positivos, es decir, las denominadas pruebas no deterministas. Por lo tanto, se propusieron varias soluciones para enfrentar estos desafíos. A continuación, se analizan las soluciones encontradas para este problema.

S9 presenta un problema relacionado con la baja estabilidad de las pruebas que interactúan con elementos de la interfaz de usuario. Reporta una mejora en la verificabilidad al proporcionar interfaces adicionales para acceder a la aplicación que se está probando mediante una API. Los pasos que se ejecutan en la etapa de precondiciones se pueden realizar utilizando API y el solo la característica particular a probar se realiza a través de la GUI. Esta solución reduce la cantidad de pasos de pruebas innecesarios en la GUI. S27 propone el mismo enfoque, donde los autores afirman que “los pasos de configuración se pueden ejecutar mediante el uso de las API REST para hacerlos más confiables y así reducir fallas innecesarias”.

En S8 se presenta un enfoque llamado “pruebas de interfaz gráfica conducidas por elementos visuales”, la cual es una técnica de pruebas que utiliza el reconocimiento de imágenes para interactuar y afirmar la exactitud de un sistema bajo prueba a través de la GUI de mapa de bits que se muestra al usuario en el monitor de la computadora. El uso del reconocimiento de imágenes permite que la técnica se use en cualquier sistema basado en GUI,

independientemente de su implementación o plataforma. A diferencia de las herramientas de pruebas basadas en GUI de segunda generación (como Selenium o Sahi), los cambios en el código de la interfaz de usuario no harán que la prueba falle. Sin embargo, se presentan varios desafíos para esta técnica en S8:

- Mantenimiento de los scripts de prueba (como en cualquier prueba automatizada de GUI).
- Sincronización entre el script de prueba y la transición del estado de la aplicación.
- Reconocimiento de imagen: se ha observado empíricamente que estas herramientas a veces no pueden encontrar la imagen esperada, produciendo falsos positivos.
- Inestabilidad.
- Falta de soporte en línea.

Por otro lado, en S37 se presentan los resultados de un caso de estudio centrado en el uso a largo plazo de pruebas de interfaz gráfica conducidas por elementos visuales en Spotify, donde mencionan los beneficios y las desventajas (Tabla 11).

En S33, los problemas de pruebas en GUI se abordan "adoptando un enfoque basado en modelos que representan los datos manipulados por la GUI y su comportamiento". Los modelos son leídos por un compilador y luego se genera código fuente Java para un “arnés de prueba”[29]. Usando el arnés de prueba, el desarrollador puede escribir casos de prueba de alto nivel, que pueden ejecutarse utilizando Selenium como controlador. Este enfoque mueve la mayoría de los factores de fragilidad del código fuente a los modelos donde manejarlos es más efectivo.

S38 propone un enfoque denominado "diferencia perceptual para Entrega Continua en aplicaciones con GUI", la cual combina conceptos de visión por computadora con Integración Continua permitiendo el reconocimiento de cambios basados en la GUI. Esto se realiza mediante comparación de imágenes. Además, proponen una herramienta que almacena capturas de pantalla de las versiones de la aplicación en producción y luego se comparan. Una vez que se generan los resultados de la comparación, el especialista en pruebas puede acceder a la herramienta para verificar su estado y aprobar o rechazar manualmente cada diferencia marcada por la herramienta. S18 presenta un enfoque de pruebas de GUI utilizando externalización abierta de tareas[30], afirmando la posibilidad de externalizar estas pruebas a un grupo de usuarios dispersos por todo el mundo.

[29] Un arnés de pruebas es el entorno de pruebas constituido por servicios que simulan sistemas externos y otros controladores (como Selenium) necesarios para ejecutar una prueba.
[30] *Crowdsourcing*

Se ha propuesto varios enfoques para enfrentar los problemas de pruebas en GUI. Sin embargo, todos tienen beneficios y desventajas que se pueden ver en la Tabla 11.

Tabla 11. Soluciones para los problemas de pruebas automatizadas de Interfaz Gráfica de Usuario.

Solución	Ventajas	Desventajas
Uso de API para las precondiciones en lugar de pasos en la GUI.	1. Velocidad en la ejecución y estabilidad de las pruebas. 2. Reducción de pruebas no deterministas. 3. Disminución de la cantidad de pasos a automatizar en la GUI.	1. Al ejecutar las regresiones, cambios en la GUI pueden seguir causando falsos positivos.
Comparación de imágenes	1. Fácil de implementar. 2. Muy preciso para detectar cambios en la GUI. 3. Útil para realizar pruebas cruzadas entre navegadores, o pruebas de compatibilidad.	1. Causa falsos positivos por diferencias en las imágenes producidos por publicidades o la manera en la que los navegadores renderizan de un modo u otro la aplicación.
Pruebas de interfaz gráfica conducidas por elementos visuales.	1. Las pruebas son fáciles de crear. 2. Flexible. 3. Se puede utilizar con cualquier sistema que utilice una GUI sin importar su implementación. 4. Cambios en el código de los elementos de la GUI no hacen que las pruebas fallen y así evita pruebas no deterministas.	1. Se necesitan actualizar las imágenes constantemente. 2. Carencia de estabilidad. 3. Poco soporte.
Enfoque basado en modelos con arnés de pruebas.	1. Flexibilidad. 2. Pruebas más rápidas y robustas. 3. Si se realizan cambios en el código fuente de la aplicación que rompe el modelo, el desarrollador recibirá un error de compilación.	1. No verifica si los elementos de la GUI se ven correctamente. 2. No se interactúa con el sistema. 3. Necesita que exista sincronización entre los casos de pruebas y la aplicación.

Solución	Ventajas	Desventajas
Externalización abierta de tareas.	1. No se necesitan automatizar las pruebas. 2. No existen pruebas no deterministas. 3. No requiere mantenimiento.	1. Dependencia de usuarios externos. 2. Los usuarios no conocen el negocio. 3. Tiempo de pruebas indeterminado.

DIFICULTADAS EN RESULTADOS DE EJECUCIÓN DE PRUEBAS AMBIGUOS

Los resultados de las pruebas deben comunicarse adecuadamente a los desarrolladores, indicando si las mismas han pasado o no. En caso de un fallo, debe quedar en claro qué fue exactamente lo que lo ha provocado. Cuando el resultado de una prueba no cumple con estos requisitos, entonces es un resultado ambiguo.

S8 presenta técnicas que automatizan el proceso de análisis de resultados. Utiliza la examinación y el análisis de archivos de volcado de memoria por fallos y archivos de registro para extraer resúmenes y detalles de fallas consistentes. Los autores de S8 llamaron a este conjunto de técnicas como "Procesamiento automatizado de resultados de pruebas".

Otro enfoque se presenta en S25. Presenta una solución que mejora la calidad de la retroalimentación con los resultados de la prueba con un reporte que incluye:

- Información sobre el conjunto de cambios introducidos que activaron la ejecución de las pruebas: registro de cambios, momento de la introducción de cambios, archivos introducidos, etc.
- Una descripción general de todas las pruebas ejecutadas con sus resultados (éxito o fallo). También se proporcionan datos sobre con qué frecuencia una prueba ya ha fallado.
- URL a diferentes páginas web con información detallada sobre las pruebas que han fallado, incluida la excepción que se ha lanzado y el informe de los elementos activos en la pila de ejecución[31].
- Un informe de cobertura de código de todas las pruebas ejecutadas para que el desarrollador pueda verificar si realmente han ejecutado el código fuente modificado y si la cobertura de código de la prueba escrita es lo suficientemente adecuada.

[31] *stack trace*

S41 reporta dos técnicas para afrontar los problemas de resultados de pruebas ambiguos:

1. Adaptación de prueba: los lotes de pruebas se segmentan y luego se adaptan según el historial de ejecuciones de pruebas. Según los autores, esto resuelve los problemas para pruebas que requieren mucho tiempo de ejecución, ejecutando las más críticas primero y otras más tarde solo si los resultados de las primeras son exitosos. También afirman que "cuando una prueba de alto nivel falla, puede ser difícil y lento descubrir por qué ocurrió la falla".
2. Pruebas individuales por confirmación de cambios: cada cambio introducido en el repositorio debe probarse individualmente, por lo que cuando las pruebas fallan, se puede detectar directamente qué cambio causó la falla.

Finalmente, S27 recomienda mejorar también los mensajes de pruebas que fallan y el nombre de las clases para las pruebas GUI hechas con Selenium. Los autores de S7 afirman que los mensajes de error exactos podrían facilitar el análisis de fallas en los resultados de las pruebas. A veces, las pruebas fallan debido a una excepción lanzada desde objetos de página[32] (patrón de *PageObject)*. Los mensajes de excepción de estos objetos de página van desde mensajes personalizados hasta excepciones detalladas de Selenium. Las excepciones de Selenium a menudo revelan la causa raíz, como, por ejemplo, cierto elemento HTML que no se encontró en la página. Sin embargo, la persona que realiza el análisis de las pruebas que fallan puede no ser capaz de reconocer el elemento por el selector de elementos web que se menciona en el error. Por lo tanto, agregar mensajes personalizados a las excepciones puede servir mejor a la claridad si son lo suficientemente precisas. El nombre de la prueba también es importante porque se usa en los resultados de las pruebas y un nombre descriptivo mejoraría en gran medida la legibilidad de estos resultados.

DIFICULTADES EN LAS PRUEBAS DE CONTENIDO DINÁMICO WEB

Las aplicaciones web modernas utilizan nuevas tecnologías como Flash, Angular o React, realizando cálculos avanzados en el lado del cliente antes de realizar una nueva solicitud de página. Además, estas solicitudes por lo general son asíncronas y probar estas tecnologías es un desafío, pero se han propuesto algunas soluciones. Para estos desafíos S5 recomienda tres actividades:

[32] PageObject es un patrón de diseño que se utiliza en pruebas automatizadas para evitar código duplicado y mejorar el mantenimiento de las mismas.

1. Probar el código de la aplicación mientras está siendo desarrollado en el servidor.
2. Probar el servidor imitando el comportamiento del cliente.
3. Probar no solo cada componente del navegador, sino también sus interacciones.

Estas actividades deben realizarse todas las noches como parte de una construcción automatizada. Dado que muchas de las pruebas en este proceso requieren la implementación de código en un servidor de producción o similar. La implementación también debe automatizarse. Además, S54 presenta una solución para sitios web con accesibilidad. La accesibilidad es un requisito de calidad no funcional para las aplicaciones web. Sin embargo, según los autores de S54, "las herramientas de evaluación automática de accesibilidad actual no son capaces de evaluar el contenido generado por DOM dinámico que caracteriza las aplicaciones Ajax y las aplicaciones de Internet enriquecidas (RIA)". En este contexto, S54 describe un enfoque para probar los requisitos de accesibilidad en RIA, mediante el uso de pruebas de aceptación. El enfoque agrega un conjunto de escenarios de usuarios de tecnología asistiva[33] a las pruebas de aceptación automatizadas, para garantizar la accesibilidad del teclado en las aplicaciones web. Estas pruebas proporcionan un análisis de accesibilidad de extremo a extremo, desde implementaciones del lado del servidor hasta del lado del cliente (JavaScript y elementos DOM generados dinámicamente) en RIA. Como las pruebas están automatizadas, se pueden incorporar en un proceso de Entrega Continua.

Aunque los dos estudios mencionados presentan enfoques para problemas basados en RIA y aplicaciones web modernas, no presentan soluciones para estos desafíos específicos de contenido dinámico. En [50], por ejemplo, se propone un marco de trabajo compuesto por Selenium y TestNG. Muestra que Selenium tiene una característica que permite que las pruebas implementen tres configuraciones diferentes de espera y tiempos de espera, para que no fallen debido al contenido dinámico. Esta característica esperará hasta que la aplicación alcance su estado final antes de continuar con el siguiente paso o verificación. Esta propuesta puede encontrarse en foros o comunidades de desarrollo como Stack Overflow, Dzone, Reddit, etc.

DIFICULTADES EN LAS PRUEBAS EN ENTORNOS BIG DATA

[33] La tecnología asistiva es cualquier dispositivo, software o equipo que ayuda a las personas a superar los desafíos para que puedan aprender, comunicarse y funcionar mejor. Por ejemplo, una silla de ruedas, un software que lee en voz alta el texto de una computadora, o un teclado para alguien que lucha con la escritura a mano.

Big Data es el proceso de manipular grandes volúmenes de datos que no pueden procesarse utilizando técnicas tradicionales. Probar el procesamiento y las características de Big Data es un nuevo desafío que involucra herramientas, técnicas y marcos de trabajo especiales.

S3 presenta dos problemas con las pruebas de Big Data y las técnicas basadas en Hadoop que pueden ser soluciones para estos:

1. "El procesamiento de Big Data lleva mucho tiempo". Una posible solución es la generación de datos de pruebas utilizando Partición de espacio de entrada con computación paralela. El proceso comienza con un modelo de dominio de entrada. Luego, el especialista en pruebas lo divide y selecciona los valores de prueba a partir de las particiones. Finalmente, se aplica un criterio de cobertura combinatoria para generar pruebas.
2. "La validación de los datos transferidos y transformados es difícil de implementar". Los datos transferidos se pueden probar verificando el número de columnas y filas, los nombres de las columnas y los tipos de datos. Si se proporcionan el origen de datos y los datos de destino, esta validación puede automatizarse. Algunos enfoques son:
    a. Validar si los datos de destino tienen tipos de datos correctos y rangos de valores a alto nivel derivando los tipos de datos y rangos de valores de los requisitos, generando pruebas para validar los datos de destino.
    b. Comparar los datos de origen con los datos de destino para evaluar si los datos de destino se transformaron o no correctamente.

En S4, se presentan técnicas de aseguramiento de calidad para aplicaciones de Big Data. Primero, los autores de S4 enumeran los atributos de calidad para aplicaciones de Big Data: precisión de datos, corrección de datos, consistencia de datos y seguridad de datos. Luego, presentan los factores de calidad de las aplicaciones de Big Data: rendimiento, confiabilidad, corrección y escalabilidad. Finalmente, discuten los métodos para garantizar la calidad de la aplicación de Big Data: arquitectura basada en modelos, monitoreo, tolerancia a fallas, verificación y predicción.

DIFICULTADES EN LAS PRUEBAS DE DATOS

Los datos son muy importantes para los diferentes tipos de sistemas y los errores en ellos son costosos y, en múltiples ocasiones, de muerte [82]. Si bien las pruebas funcionales de software han recibido mucha atención, las pruebas de datos han sido pasadas por alto.

S41 informa una solución parcial para problemas de pruebas de datos usando pruebas de “cambios de esquema de base de datos”. En S19, los autores proponen un enfoque llamado

“Pruebas Continuas de Datos” que ejecuta consultas de pruebas en segundo plano, mientras que un desarrollador o administrador de base de datos modifica una base de datos (BD). Esta técnica notifica al usuario sobre errores de datos cuando se introducen, lo que genera tres beneficios:

1. El error se descubre rápidamente y se puede solucionar antes de que cause un problema.
2. Aumenta la posibilidad de corregirlo rápidamente.
3. Contribuir a escasa documentación de datos.

Según los autores de S19, "las Pruebas Continuas de Datos pueden descubrir múltiples tipos de errores, en los que se incluyen errores de corrección y errores que degradan el rendimiento". Concluyen que "el objetivo no es detener la introducción de errores, sino acortar el tiempo de detección tanto como sea posible".

S49 presenta un enfoque llamado TDD para bases de datos relacionales. Para extender la práctica de TDD al desarrollo de bases de datos, son necesarias tareas de base de datos equivalentes a pruebas de regresión, refactorización e Integración Continua:

- En las pruebas de regresión de bases de datos, la base de datos se valida ejecutando un conjunto completo de pruebas de caja negra y caja blanca.
- En la refactorización de bases de datos, se realiza un cambio simple en una base de datos que mejora su diseño (manteniendo su semántica conductual e informativa).
- En la Integración Continua de BD, los desarrolladores integran sus cambios en sus instancias de bases de datos locales, incluidos los cambios estructurales, funcionales e informativos. En cualquier caso, cada vez que alguien envíe un cambio en la base de datos, las pruebas deben verificar que la base de datos se mantenga estable. Después de eso, todos los demás que trabajan con la misma base de datos deben descargar el cambio del sistema de control de versiones y aplicarlo a su propia instancia local de la base de datos lo antes posible.

  Los autores de S49 presentan buenas prácticas para la Integración Continua de BD:

- Automatizar la construcción.
- Colocar todo en el repositorio de control de versiones (scripts de datos, esquemas de bases de datos, datos de prueba, modelos de datos y artefactos similares).
- Brindar a los desarrolladores sus propias copias de la base de datos.

DIFICULTADES EN LA IMPLEMENTACIÓN DE PRUEBAS EN DISPOSITIVOS MÓVILES

Las pruebas en dispositivos móviles han traído consigo muchos desafíos con respecto al proceso de pruebas, los artefactos de pruebas, los niveles de pruebas, los diferentes tipos de pruebas, los tipos de dispositivos y los costos de las pruebas automatizadas. Debido a estos desafíos, han surgido algunas propuestas para pruebas en dispositivos móviles.

S13 presenta el proceso de Despliegue Continuo de software de la aplicación móvil de Facebook. En ese artículo, los autores mencionan que las pruebas son particularmente importantes para las aplicaciones móviles porque:

1. Muchas actualizaciones de software para dispositivos móviles se realizan cada semana.
2. Hay muchos dispositivos y sistemas operativos donde el software debe ejecutarse.
3. Cuando surgen problemas críticos en producción hay pocas opciones para tratarlos.

Según S13, "Facebook aplica numerosos tipos de pruebas, incluidas pruebas unitarias, pruebas de análisis estático de código fuente, pruebas de integración, pruebas de diseño de pantalla, pruebas de rendimiento, pruebas de compilación y pruebas manuales". Estas pruebas son automáticas y se ejecutan en cientos de nodos utilizando entornos simulados y emulados. Para las pruebas de rendimiento en hardware real, Facebook utiliza un laboratorio de dispositivos móviles, haciendo foco principalmente en las características de la aplicación, como la velocidad, el uso de la memoria y la eficiencia de la batería. El laboratorio de dispositivos móviles contiene racks aislados electromagnéticamente. Cada rack contiene múltiples nodos que están conectados a dispositivos móviles reales con diferentes sistemas operativos. Por lo tanto, la estrategia de prueba de Facebook abarca los siguientes principios:

- Cobertura: las pruebas deben ejecutarse tan exhaustivamente como sea posible.
- Velocidad: las pruebas de regresión deben ejecutarse en paralelo. El objetivo es proporcionar al desarrollador los resultados de las pruebas de humo dentro de los 10 minutos de la introducción de los cambios al repositorio principal.
- Confiabilidad: las pruebas deben identificar problemas con precisión. Las pruebas no deterministas deben minimizarse.
- Automatización: se deben automatizar tantos casos de pruebas como sea posible.
- Priorización: las pruebas deben priorizarse ya que utilizan demasiados recursos informáticos en las regresiones.

S35 propone un marco de trabajo que se puede aplicar para probar diferentes navegadores y aplicaciones móviles utilizando una herramienta llamada Appium que funciona para aplicaciones web nativas, híbridas y móviles para sistemas iOS y Android. Es un marco de automatización de pruebas basado en datos que utiliza la biblioteca de pruebas Appium.

Asimismo, su principal funcionalidad es probar de forma automática las aplicaciones móviles. Según un estudio realizado por los autores de S35, "el marco de trabajo mejorará el proceso de prueba para aplicaciones móviles, ahorrará tiempo y acelerará el proceso de pruebas, permitiendo además desplegar en producción cualquier aplicación móvil en un pequeño período de tiempo".

Finalmente, S20 presenta una combinación de Appium y TaaS. Los autores de S20 describen un caso de estudio de la herramienta MedTabImager que utiliza este marco de trabajo para ejecutar pruebas de GUI. Estas pruebas se ejecutaron contra un servicio en la nube (Sauce Labs). Según los autores de S20, "configurar Sauce Labs en el servidor Integración Continua con un complemento específico no requirió mucho esfuerzo". Una construcción de la aplicación exitosa (que incluye pruebas unitarias) activa las pruebas de GUI en Sauce Labs automáticamente. Aunque las pruebas están automatizadas, se requieren ciclos de pruebas manuales antes de la publicación. Para automatizar la distribución de la aplicación, S20 utilizó "distribución de compilaciones beta" utilizando un servicio basado en la nube llamado TestFairy4. Para MedTabImager, los autores afirman que "la solución TaaS requiere menos esfuerzo de configuración y funciona bastante bien".

Todos los niveles y etapas de pruebas discutidos para la **cuestión 2** se pueden incorporar para pruebas móviles. Los casos de pruebas para aplicaciones móviles se pueden automatizar utilizando herramientas existentes como Appium. También se pueden paralelizar y existen tres enfoques para entornos de pruebas móviles:

1. Dispositivos emulados con diferentes sistemas operativos, para entornos de desarrollo.
2. Dispositivos móviles físicos: para entornos similares a la producción.
3. Servicio en la nube (como Sauce Labs)

DIFICULTADES EN PRUEBAS AUTOMATIZADAS DE REQUERIMIENTOS NO FUNCIONALES

Si bien las pruebas unitarias, de integración y funcionales se han discutido ampliamente en la literatura y se han practicado ampliamente en Entrega Continua, las pruebas de requerimientos no funcionales se han pasado por alto.

En la Tabla 12, se presenta una lista de los diferentes requerimientos no funcionales que fueron considerados por los estudios en la implementación de los conductos de despliegue. La Fig. 24 también muestra la cantidad de veces que se consideraron.

Tabla 12. Pruebas en requerimientos no funcionales implementados en Entrega Continua.

Requerimiento no funcional	Herramienta – Técnica – Enfoque	Artículo
Mantenibilidad	SonarQube, Gerrit	S2, S13, S22, S47, S9, S11, S39, S12
Rendimiento	JMeter	S2, S13, S47, S43, S9, S11, S39
Correcta instalación y/o despliegue	JUnit, TestNG, XUnit, NUnit	S14, S22
Compatibilidad	Pruebas Cross-Browser	S14, S39
Localización e internalización	Pruebas G11N/L10N	S14
Carga	JMeter, Grinder, Gatling	S14, S43, S9, S11
Sobrecarga (Stress)	JMeter, Gatling	S14, S9
Documentación	Solución Propia	S14
Seguridad	Solución Propia	S22
Accesibilidad	Solución Propia	S54
Usabilidad	Solución Propia	S22

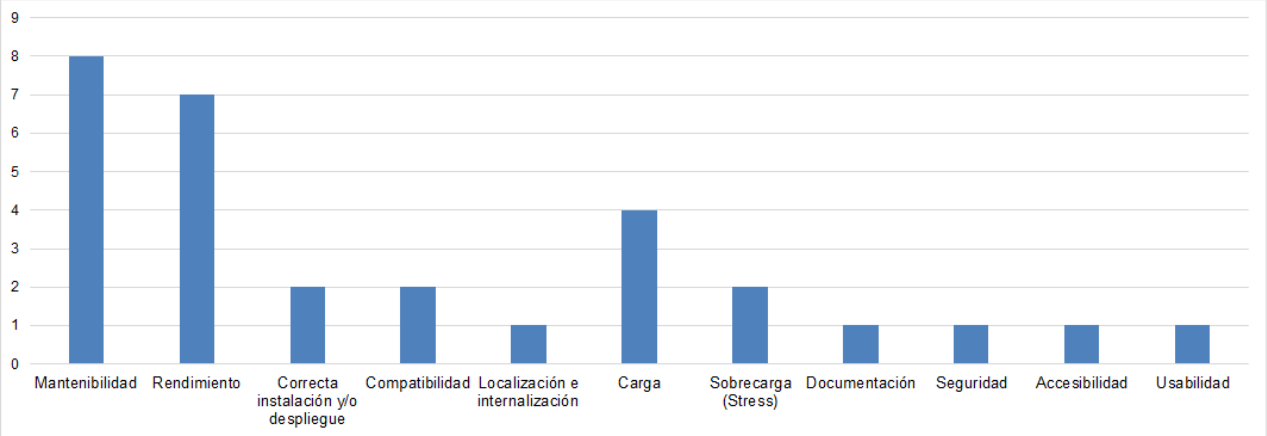


Fig. 24. Requerimientos no funcionales en Entrega Continua.

3.2.2.4. Cuestión 4: ¿Existen todavía problemas de pruebas en proyectos de desarrollo continuo?

Después del análisis de los 56 artículos, se encuentran diferentes desafíos y problemas existentes en el proceso de pruebas de Entrega Continua. Estos desafíos se enumeran a continuación:

- Pruebas Continuas de aplicaciones compuestas por servicios en la nube (S6): la cantidad de servicios en la nube disponibles y el tamaño de los datos que necesitan manejar a menudo son grandes. Probar flujos de trabajo compuestos por esos servicios en la nube es un desafío.
- Monitoreo continuo (S12, S53): con la capacidad de ejecutar pruebas tempranamente en un sistema similar a producción, existe la oportunidad de monitorear varios parámetros de calidad y, por lo tanto, la capacidad de reaccionar ante problemas repentinos de manera oportuna.
- Desafíos en pruebas de GUI (S28, S37): mantenimiento de los scripts de pruebas, sincronización entre el script de prueba y la transición del estado de la aplicación, falsos positivos en los resultados de prueba e inestabilidad.
- Pruebas como un servicio (TaaS) en Entrega Continua (S26): Los servicios más populares son SaurceLabs, BlazeMeter y SOASTA CloudTest. BlazeMeter presentó en 2015 una solución para la Entrega Continua utilizando TaaS (Pruebas Continuas como servicio[34]).
- Pruebas automatizadas de microservicios (S32): Según los autores de S32, el concepto de microservicios es relativamente nuevo, por lo que hay pocos artículos en el campo de la validación y prueba de microservicios en este momento.

DISCUSIÓN DE LA CUESTIÓN 4

[34] https://www.blazemeter.com/blazemeter-news/blazemeter-introduces-continuous-testing-service-marketplace.

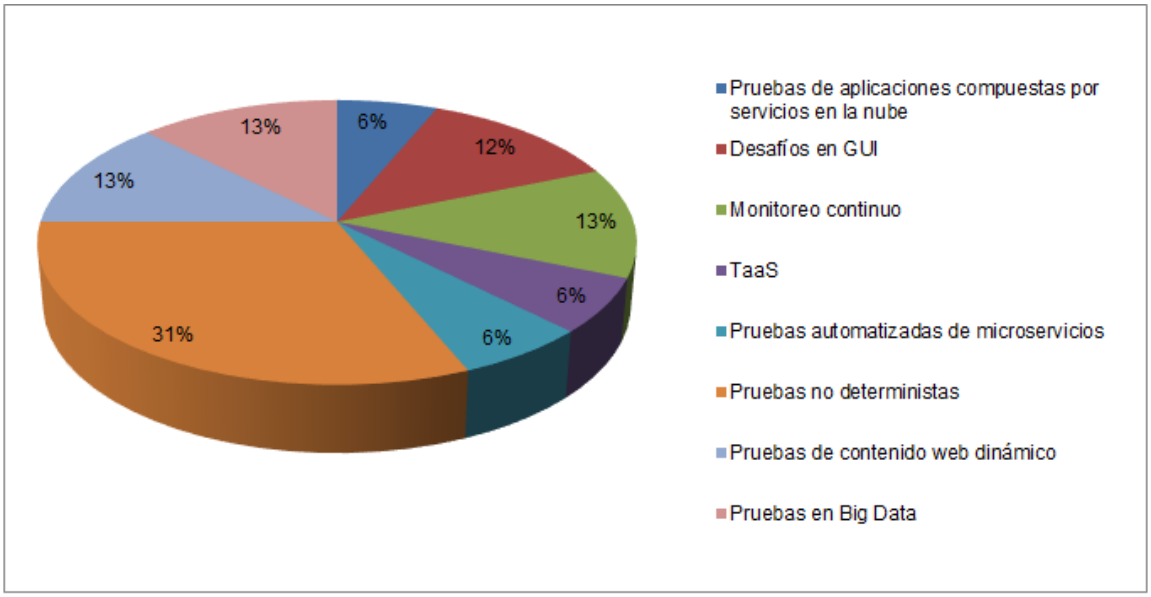


Fig. 25. Problemas existentes relacionados a las pruebas en Entrega Continua.

Si bien se encontraron problemas en el campo de Pruebas Continuas, algunos de ellos mencionados anteriormente en este artículo no se han aún resuelto completamente. Por esa razón, se han agregado esos problemas sin resolver a la lista de problemas existentes. Esto se puede ver en la Fig. 25.

3.2.3. Conclusiones de la RSL

El primer objetivo fue validar si existía una definición aceptada de Pruebas Continuas (PC) y si estaba relacionada con los otros enfoques continuos (IC, DC, EC). Los resultados muestran que el concepto de Pruebas Continuas ha evolucionado a lo largo de los años. Al principio, solo se aplicaba a la ejecución de pruebas unitarias de forma continua y ahora no se aplica solo a las pruebas unitarias, sino también a cada tipo de caso de prueba que se puede automatizar. Por lo tanto, se validó que Pruebas Continuas es el proceso de ejecutar cualquier tipo de caso de prueba automatizado lo más rápido posible para proporcionar retroalimentación rápida a los desarrolladores y detectar problemas críticos antes de pasar a producción. Esta definición de Pruebas Continuas es uno de los objetivos principales de Entrega Continua, por lo que se concluyó que PC está directamente relacionado con los entornos de desarrollo continuo.

También se buscaron diferentes niveles o etapas de pruebas en Entrega Continua. Las pruebas unitarias, las pruebas funcionales y las pruebas de rendimiento, carga y estrés son las etapas más utilizadas en entornos de desarrollo continuo de software.

El tercer objetivo consistía en analizar propuestas, técnicas, enfoques, herramientas y otro tipo de soluciones para los diferentes problemas de prueba existentes en Entrega Continua. Se han propuesto muchas soluciones para enfrentar los problemas mencionados. Las pruebas que llevan mucho tiempo han sido el problema más discutido en los artículos, donde se han propuesto nuevas técnicas, enfoques y herramientas para resolverlo. Por otro lado, las pruebas no deterministas, las pruebas de Big Data y las pruebas de aplicaciones web modernas que usan Flash, Angular, React o tecnologías similares, realizando llamadas asíncronas son problemas que aún no se han resuelto por completo. Sin embargo, las diferentes soluciones detectadas pueden significar una contribución a Entrega Continua y a partir de ellas se puede crear una lista de factores clave de éxito.

Finalmente, se intentaba detectar la presencia de problemas aún existentes para las pruebas en Entrega Continua. Se generó una lista de 5 problemas sin solución: Pruebas Continuas de aplicaciones basadas en servicios en la nube, desafíos con pruebas de GUI, monitoreo continuo, TaaS en Entrega Continua y pruebas automatizadas de microservicios. Sin embargo, a esa lista se agregaron las pruebas no deterministas, las pruebas de Big Data y las pruebas de contenido web dinámico, ya que aún siguen siendo problemas sin resolver.

3.3. Análisis de los problemas encontrados en la literatura

Tomando los resultados de la RSL descritos en la sección 3.2, se realizó un análisis de los problemas detectados [14]. Este análisis contempla todos los problemas que han sido reportados por diferentes autores en la literatura, relacionándolos para determinar las causas principales de su existencia en Pruebas Continuas.

De acuerdo a los resultados encontrados en la RSL [11] y el análisis realizado [14], los problemas más reportados son:

- **Pruebas que consumen mucho tiempo de ejecución**: existen muchos tipos de pruebas que pueden implementarse en Entrega Continua. Sin embargo, ejecutar un gran lote de pruebas es una tarea que lleva mucho tiempo. Además, los cambios son introducidos con más frecuencia y por eso, es necesario la ejecución de regresiones lo más rápido posible.
- **Pruebas no deterministas**: Una prueba no determinista es aquella que podría generar un resultado positivo o negativo por la misma versión del software. Son pruebas que producen los llamados “falsos positivos” y por su inestabilidad son más conocidas como

pruebas no deterministas. Una de las características más importantes de Entrega Continua es la confiabilidad y pruebas que fallan aleatoriamente no son confiables.

- **Automatización de pruebas de interfaz gráfica de usuario**: Las pruebas de interfaz de usuario eran ejecutadas de manera manual, pero con la aparición de herramientas como Selenium WebDriver, fue posible comenzar a automatizarlas. Sin embargo, la interfaz de usuario es la parte de la aplicación que cambia con más frecuencia y realizar pruebas sobre la misma, puede ocasionar el no determinismo en las pruebas.
- **Resultados de ejecución de pruebas ambiguos**: En entornos de desarrollo continuo, los desarrolladores deben ser notificados de cualquier defecto introducido, detectado a través de las pruebas. Cuando los resultados de la ejecución no son claros, es decir, no detallan el error, su causa, el lugar donde ocurrió, capturas de pantalla, etc., se tienen resultados ambiguos.
- **Automatización de pruebas web con contenido dinámico**: Las aplicaciones web más modernas, utilizan tecnologías como React o Angular, que generan contenido dinámico. Incorporar pruebas automatizadas de este tipo de tecnologías en un Conducto de Despliegue es complejo, ya que una determinada página web pudo haber alcanzado el estado esperado final, pero el contenido dinámico aún se encuentra cargando.
- **Pruebas en Big Data**: Big Data es el proceso de utilizar grandes conjuntos de datos que no se pueden procesar utilizando técnicas tradicionales. Realizar verificaciones sobre este conjunto de datos es un nuevo reto que involucra varias técnicas y herramientas que aún están madurando y por lo tanto es difícil incorporarlas en Entrega Continua.
- **Pruebas de datos**: Los datos son muy importantes para diferentes tipos de sistemas y los errores en estos son costosos. Si bien las pruebas de software han recibido gran atención en diferentes niveles, las pruebas de datos han sido poco tenidas en cuenta. Es por ello, que no existen soluciones totales, para la implementación de pruebas de datos en un entorno de Entrega Continua.
- **Pruebas en dispositivos móviles**: Las pruebas móviles han traído consigo muchos desafíos. Entre ellos se encuentran: el proceso de pruebas en sí, los niveles y tipos de pruebas a considerar, los diferentes tipos de dispositivos y los costos de las pruebas automáticas. Todos estos factores dificultan la implementación de pruebas en dispositivos móviles en Entrega Continua.
- **Pruebas automatizadas de requerimientos no funcionales**: Si bien las pruebas unitarias, de integración y funcionales, han sido estudiadas ampliamente en la literatura e

implementadas por muchas organizaciones en la práctica, las pruebas no funcionales han sido poco consideradas.

- **Pruebas de aplicaciones compuestas por servicios en la nube**: Actualmente, existe una gran cantidad de servicios en la nube y el tamaño de los datos que deben manejar es muy grande. Probar flujos compuestos por estos servicios es muy complejo. Además, es necesario proporcionar garantías de calidad de servicio [32].
- **TaaS en Entrega Continua**: Las pruebas como un servicio es un modelo en cual las pruebas son ejecutadas por un proveedor de servicios en lugar de un equipo de pruebas perteneciente a la misma organización que desarrolla el software. Realizar TaaS en Entrega Continua, resulta una tarea muy compleja, puesto que los componentes y procesos del proveedor de servicios deben adaptarse a las características del Conducto de Despliegue de la organización contratante.
- **Pruebas de servicios web**: Si bien existen muchas herramientas para probar este tipo de arquitecturas, es difícil integrarlas en Entrega Continua.

Los problemas mencionados representan dificultades para la implementación del proceso de pruebas en Entrega Continua correctamente. Además, los mismos se encuentran relacionados entre sí, de tal manera, que la ocurrencia de uno produce directamente otro, o lo puede producir. Estas relaciones se muestran en la Fig. 26, donde puede apreciarse que los problemas más importantes son las **pruebas que consumen mucho tiempo de ejecución** y las **pruebas automatizadas no deterministas**.

Además, en base a los problemas relacionados con dificultades en la implementación de ciertos tipos de pruebas, se ha agregado un problema más: **poca cobertura de pruebas en Entrega Continua**, debido a la carencia de herramientas, marcos de trabajo, modelos y/o conjunto de buenas prácticas para pruebas automatizadas. Estos factores han sido impedimentos para algunas organizaciones en la adopción de EC.

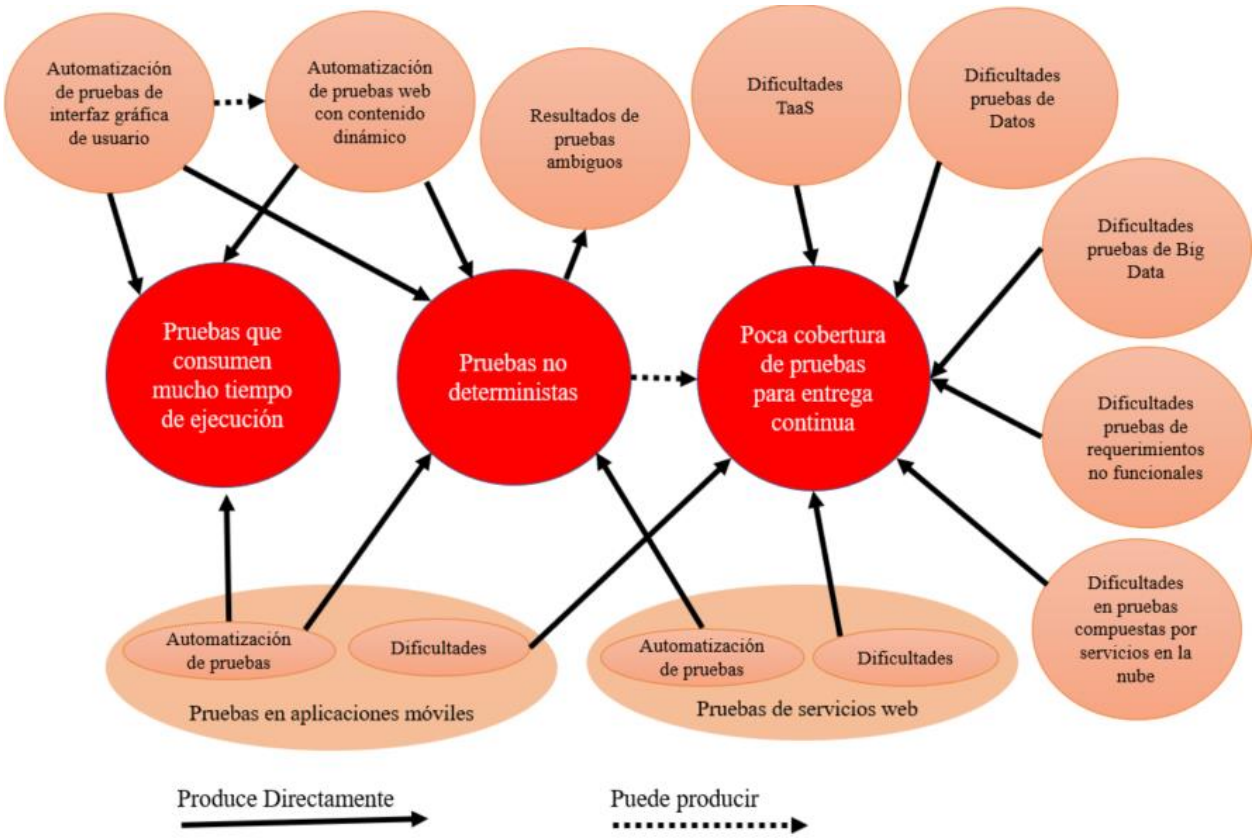


Fig. 26. Relación entre los problemas de pruebas en Entrega Continua [14].

3.4. Los problemas de Pruebas Continuas en la industria

Como se menciona en la sección 3.3, los problemas reportados en la literatura son:

- Pruebas que consumen mucho tiempo de ejecución
- Pruebas no deterministas
- Automatización de pruebas de interfaz gráfica de usuario
- Resultados de ejecución de pruebas ambiguos
- Automatización de pruebas web con contenido dinámico
- Pruebas en Big Data
- Pruebas de datos
- Pruebas en dispositivos móviles
- Pruebas automatizadas de requerimientos no funcionales
- Pruebas de aplicaciones compuestas por servicios en la nube
- TaaS en Entrega Continua
- Pruebas de servicios web

Asimismo, al realizar un análisis y relación entre estos problemas, se detectó que los más importantes son las pruebas que consumen mucho tiempo de ejecución, las pruebas automatizadas no deterministas y la escasa cobertura para algunos niveles de pruebas, debido a falta de documentación, modelos, marcos de trabajo, buenas prácticas, patrones, etc. Sin embargo, esos resultados se extrajeron solo de la literatura, utilizando fuentes académicas que no están directamente relacionadas con la industria. Por lo tanto, es necesario validar si la industria enfrenta los mismos problemas, si nunca los han experimentado o si ya los han resuelto.

Para ello, con el objetivo de buscar, identificar y proporcionar información sobre el estado de los procesos de pruebas en proyectos que están desarrollando diferentes tipos de software, utilizando Integración Continua y prácticas similares como Despliegue Continuo o Entrega Continua, se llevó a cabo una encuesta [15]. En dicha investigación, se buscaba validar si todavía existían desafíos o problemas sin solución y relevar las soluciones que se pueden tomar como base para la creación de nuevas herramientas y marcos de trabajo.

3.4.1. Encuesta sobre Pruebas Continuas

Esta investigación busca identificar y proporcionar información sobre el estado de los procesos de prueba en proyectos que están desarrollando diferentes tipos de software con enfoques de desarrollo continuo. El objetivo principal es detectar los problemas relacionadas a las pruebas en estos entornos y las soluciones que pueden tomarse como base para la creación de nuevas herramientas y procedimientos. Para ello, se empleó una encuesta como método de recopilación de datos. En lo que respecta al diseño y desarrollo de la encuesta, se siguieron los principios de Kitchenham de investigación con encuestas [37], [134]–[138]. A continuación, se describe el procedimiento seguido para el diseño y la realización del estudio.

3.4.1.1. Objetivos, metodología y diseño de la encuesta

El primer paso para comenzar esta investigación fue establecer las metas derivadas de los estudios previos. En el contexto del desarrollo continuo de software en la industria, los objetivos son:

- Identificar si la industria se enfrenta a los mismos problemas de pruebas reportados por la literatura y determinar su gravedad.
- Buscar otros desafíos y problemas aún no resueltos al implementar el proceso de pruebas.

- Evaluar el uso de prácticas de desarrollo continuo que puedan contribuir a la calidad de los procesos y productos de software.
- Detectar y relevar soluciones para los problemas y desafíos reportados.

Partiendo de los objetivos mencionados, se comienza a diseñar la encuesta utilizando documentos relacionados y estudios similares. El tipo de encuesta es transversal [38], porque se buscaba obtener el panorama de las pruebas y prácticas relacionadas con el desarrollo continuo en el presente. El instrumento de encuesta fue el cuestionario y luego de concluir con el diseño del mismo, se llevó a cabo un proceso piloto para evaluar la validez y legibilidad de las preguntas. En esta etapa, cuatro gerentes de proyecto y diez desarrolladores de software expertos proporcionaron recomendaciones sobre cómo mejorar la encuesta. Se abordaron los comentarios provistos y se mejoró la encuesta. Luego, se repitió el proceso con tres gerentes de proyecto más, donde luego de aplicar las mejoras, se obtuvo el cuestionario final.

La población objetivo estaba constituida por proyectos y equipos que trabajan bajo metodologías ágiles e implementan prácticas como Integración Continua, Despliegue Continuo o Entrega Continua. Sin embargo, como la población objetivo es extremadamente grande, tomando otros trabajos similares que tienen un tamaño de muestra en un rango de entre 100 y 250 encuestados [54], [139]–[143], el tamaño de muestra para esta encuesta fue de 255 proyectos. Los mismos pertenecen a empresas ubicadas principalmente en los Estados Unidos, Canadá, Inglaterra, Alemania, Suiza, Argentina, Brasil, Chile, Irlanda y China. Los proyectos son de desarrollo de aplicaciones web de aerolíneas, sitios de comercio electrónico, sitios de agencias de viajes, motores de búsqueda, compañías de software de relaciones públicas, compañías de seguros, bancos, portales de noticias, entre otros. Se utilizaron las redes sociales empresariales (como por ejemplo LinkedIn) para invitar a las personas a completar la encuesta. Además, se proporcionó una breve introducción en la que los encuestados pudieron ver el propósito del estudio, la importancia de su participación y cómo se beneficiarían de él. Como dice Kitchenham [136], "las personas estarán más motivadas para proporcionar respuestas completas y precisas si pueden ver que los resultados del estudio probablemente les sean útiles".

Había tres secciones en la encuesta. La primera sección fue sobre información del proyecto, como el número de miembros del equipo, sus roles, el tipo de aplicación que se está desarrollando y la duración del tiempo del proyecto. Esa información se utiliza como metadatos y es importante en el momento de la discusión. La segunda sección fue sobre el uso de prácticas de desarrollo continuo. Para medir el uso de estas prácticas, se hicieron preguntas que determinan cómo se implementan las buenas prácticas y patrones de IC, DC y EC de acuerdo con [3], [4], [7], [9], [10]. La última sección fue sobre problemas y soluciones en el proceso de

prueba. La mayoría de las preguntas de esta sección eran preguntas abiertas, de modo que los encuestados pudieron expresar todos los problemas que enfrentan y las soluciones o soluciones parciales que pudieron implementar.

3.4.1.2. Resultados de la encuesta

Como se mencionó anteriormente, la encuesta se componía de 3 partes: información del proyecto, uso de prácticas de desarrollo continuo, y, problemas y soluciones de pruebas en ambientes continuos. El detalle referente a los proyectos participantes y el cuestionario utilizado para recabar la información se presentan en el ANEXO B.

USO DE PRÁCTICAS DEL DESARROLLO CONTINUO DE SOFTWARE

Como se mencionó anteriormente, para medir el uso de estas prácticas, se hicieron preguntas que permiten determinan cómo se implementan las mejores prácticas y patrones de IC, DC y EC de acuerdo con [1] - [4], [33]. La primera parte de esta sección fue sobre el uso de buenas prácticas por el desarrollador de forma local, donde se queria determinar si los miembros del equipo están construyendo y probando el código correctamente antes de integrarlo en la rama principal del repositorio de control de versiones y también, la frecuencia de estas integraciones. Como se puede ver en el cuestionario detallado en el ANEXO B, las preguntas para esta parte son Q2.1, Q2.2, Q2.3 y Q2.4. Los resultados se presentan en la Fig. 27.

De los 255 proyectos, 15 de ellos (6%) manifestaron que ocasionalmente compilan el código antes de integrarlo a la rama principal en el repositorio de control de versiones, ya que los cambios pequeños se integran directamente. Todos afirmaron que solo compilan/construyen el código cuando los cambios en el mismo son muy grandes. El caso de las pruebas unitarias es bastante similar: 29 de los encuestados (12%) declararon que solo ejecutan pruebas unitarias localmente (o en una rama separada) cuando los cambios en el código son grandes. Con respecto a los 24 proyectos que a veces ejecutan un conjunto de pruebas de aceptación automatizadas antes de integrar el código (9%), el problema principal es el tiempo. Según ellos, ejecutar pruebas automatizadas lleva mucho tiempo. Por lo tanto, algunos de los proyectos ejecutan las pruebas automatizadas solo cuando se trata de un cambio de código muy importante. Otros proyectos ejecutan pruebas automatizadas específicas relacionadas con la funcionalidad que se está modificando, pero depende del tiempo disponible y si esas pruebas automatizadas relacionadas pueden determinarse.

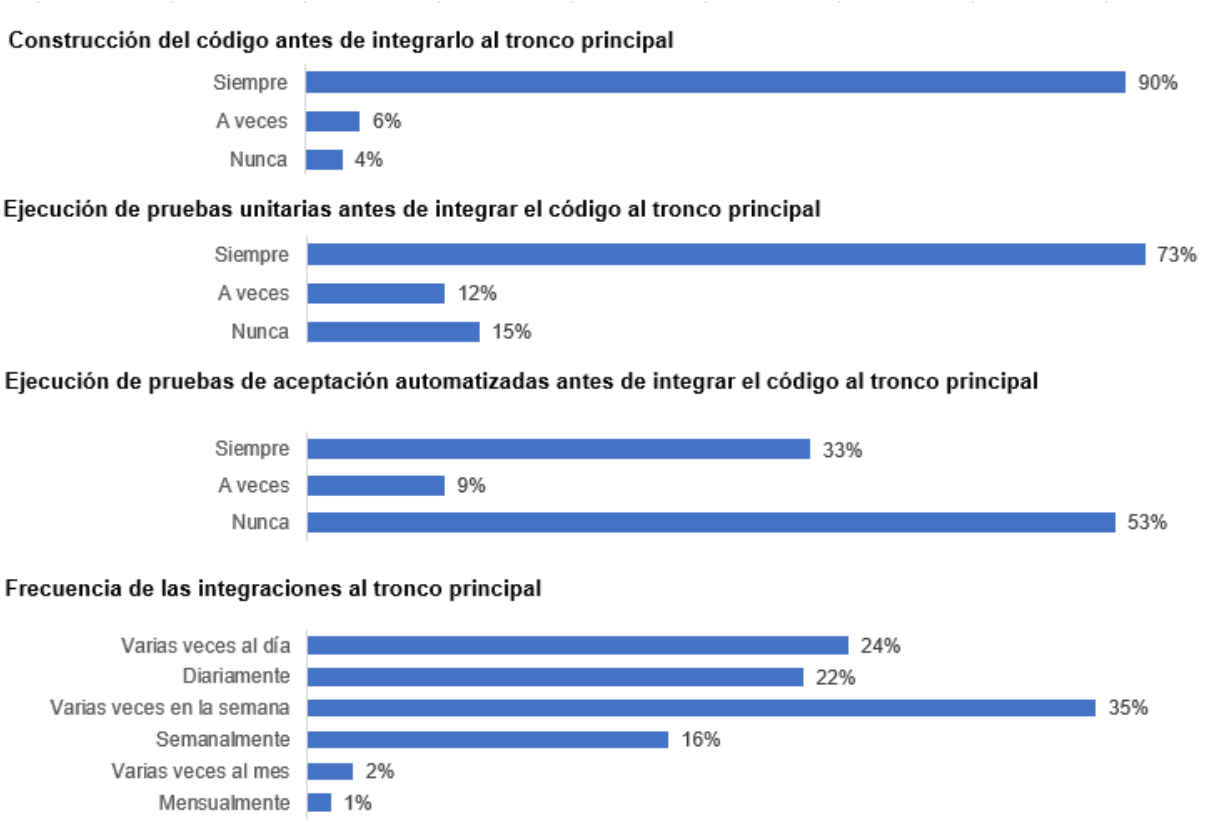


Fig. 27. Uso de prácticas en los entornos locales de los desarrolladores.

Finalmente, la Tabla 13 resume la frecuencia de las integraciones a la rama principal.

Tabla 13. Integraciones a la rama principal.

Frecuencia	Cantidad de proyectos
Muchas veces al día	63
Una vez al día	57
Muchas veces a la semana	89
Una vez a la semana	42
Muchas veces en el mes	3
Una vez al mes	1

Como se mencionó anteriormente, todos los proyectos utilizan una o más prácticas de desarrollo continuo. Por lo tanto, el código sigue un curso compuesto de diferentes etapas [1] que implican construir, probar y desplegar el software.

La Fig. 28 muestra las diferentes etapas utilizadas por los proyectos en sus flujos y la manera en que se activan o disparan. En el cuestionario presentado en el ANEXO B, las preguntas para estos resultados son Q2.6 y Q2.7.

Fig. 28. Modos de activación del flujo de integración y sus etapas.

Con respecto a los modos de activación de flujos de integración del código, hubo varios proyectos que seleccionaron más de una opción porque todos han manifestado tener 2 flujos diferentes: uno para la integración y construcción del código base, con pruebas unitarias, mientras que el otro flujo es utilizado para las pruebas de aceptación automatizadas. Además, hay algunos proyectos que activan el flujo de integración automáticamente con el solo hecho de crear solicitudes[35] de integración de código.. La Tabla 14 muestra los detalles de los modos de activación de flujo de integración de código.

Tabla 14. Modo de activación para flujos de integración por proyecto.

Modo de activación para flujos de construcción y pruebas unitarias	Modo de activación para flujos de pruebas de aceptación automatizadas	Cantidad de proyectos
Automatizado	Automatizado	67
Programado	Programado	43
Manual	Manual	75
Automatizado	Programado	5
Automatizado	Manual	24
Programado	Manual	32
Otro	Otro	9

[35] *Pull requests*

Duvall [9] y Fowler [10] afirman que Integración Continua es crucial para lograr Despliegue Continuo o Entrega Continua. En el libro de Duvall et al. [9], se mencionan un conjunto de buenas prácticas para implementar Integración Continua adecuadamente, las cuales se enumeran en la sección 2.3. En la encuesta realizada, se ha cuestionado a los proyectos acerca del uso de esas prácticas. En el ANEXO B, estas preguntas son de Q2.8 a Q2.20. La Fig. 29, la Fig. 30, la Fig. 31 y la Fig. 32 muestran los resultados.

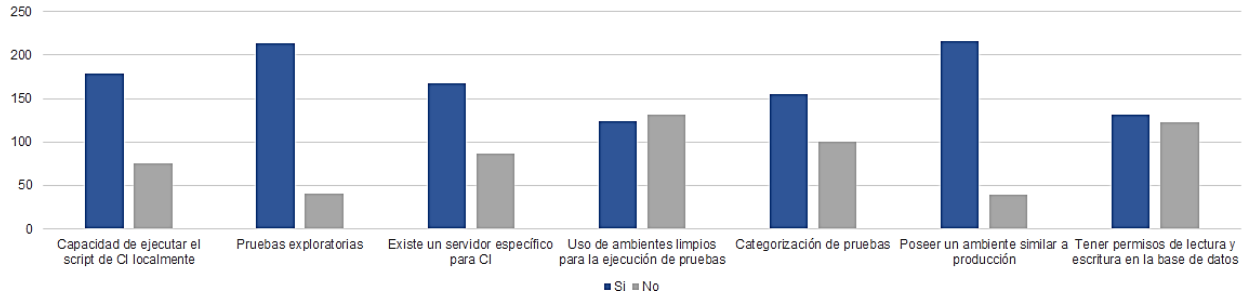


Fig. 29. Uso de las principales buenas prácticas de Integración Continua.

Según Duvall [9], todos los desarrolladores deben llevar a cabo compilaciones y construcciones privadas en sus estaciones de trabajo locales antes de enviar su código al repositorio de control de versiones. Eso asegurará que sus cambios no causarán errores en el flujo de integración. Para hacerlo, deberían poder ejecutar localmente el mismo script que ejecutan en el servidor de Integración Continua. De los 255 proyectos, 179 pueden realizar esta tarea y 76 no. Además, se debe utilizar una máquina de integración separada para ejecutar las etapas de este flujo de integración. 168 de los proyectos están usando una máquina separada para sus servidores de IC y 87 están usando máquinas para IC y otros fines. Además, 124 proyectos utilizan entornos limpios para ejecutar sus pruebas automatizadas y 131 no.

Con respecto a las buenas prácticas de control de calidad, el 84% de los proyectos (214) están ejecutando pruebas exploratorias para nuevas características y el 61% (155) categorizan las pruebas. Además, es importante contar con un entorno similar a producción para las pruebas, de modo que los riesgos relacionados con la introducción de errores en producción sean bajos. La mayoría de los proyectos (216) declararon que tienen un entorno similar a producción y solo 39 de ellos no lo tienen.

Según Humble y Farley [7], para implementar Entrega Continua, la colaboración entre los diferentes miembros del equipo es muy importante y Duvall [9] afirma que todos deberían tener acceso a las bases de datos. Los resultados de la encuesta mostraron que casi la mitad de los proyectos tienen desarrolladores o especialistas en pruebas que no tienen permisos de lectura y escritura para las bases de datos (48%).

Fowler afirma que "una parte clave de hacer una construcción de código continua es que, si la construcción falla, debe repararse de inmediato" y que generalmente, "la forma más rápida de solucionarla es quintando los últimos cambios de la rama principal en el repositorio de control de versiones, llevando al sistema de vuelta a la última versión estable" [4].

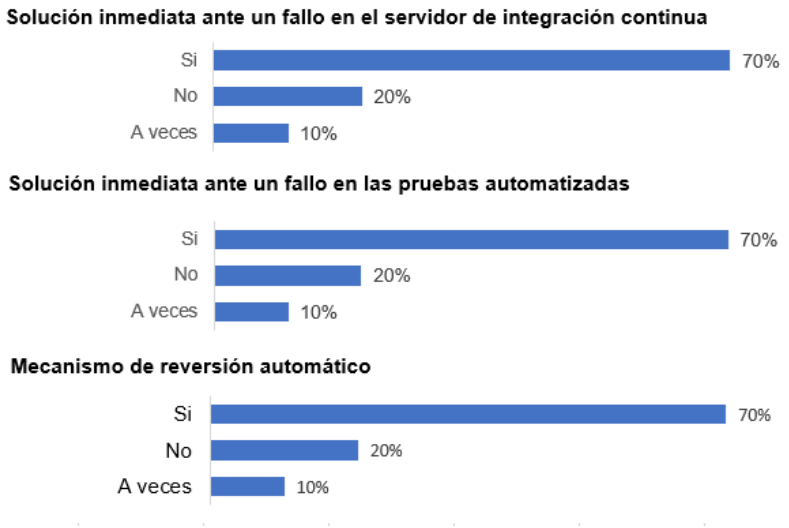


Fig. 30. Prácticas de manejo de fallas en los flujos de Integración Continua.

Los resultados de la encuesta muestran que 178 de los proyectos (70%) comparten esta filosofía, donde la tarea de mayor prioridad es arreglar un error en el flujo del servidor de Integración Continua. Además, otros 26 proyectos (10%) manifestaron que a veces arreglan la construcción de inmediato dependiendo de la situación. Todos expresaron que, si se trata de un problema crítico, revertirán los cambios en el código, de lo contrario lo ignorarán. Finalmente, 51 proyectos (20%) no arreglan el flujo de IC inmediatamente después de una falla. La mayoría de ellos declararon que simplemente lo reportan como un error o problema en sus herramientas de seguimiento de incidentes, para que otro equipo o persona pueda solucionarlo más tarde. En cuanto al mecanismo para revertir los últimos cambios, 133 proyectos lo tienen y 114 no lo tienen. En el mismo contexto, como hubo muchos proyectos que declararon que tienen flujos de Integración Continua separados, uno para la integración y construcción del código base con pruebas unitarias y otro para las pruebas de aceptación automatizadas, se agregó una pregunta para ver si también se ocupan de las pruebas automatizadas al fallar en el servidor tan pronto como aparecen. Los resultados muestran que 153 de los proyectos lo hacen y 102 no abordan esos problemas.

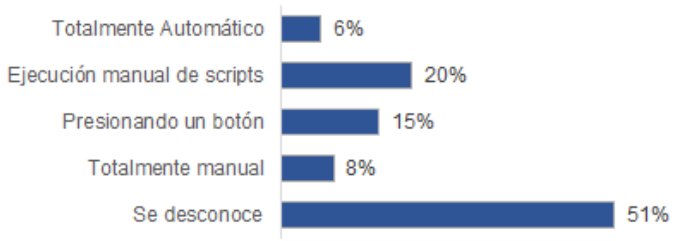


Fig. 31. Modos de despliegue a producción.

El mecanismo que utilizan los equipos para liberar el software que desarrollan a producción también es importante. Despliegue Continuo se trata de desplegar en producción automáticamente después de que las etapas anteriores se completen con éxito. Por otro lado, Entrega Continua se trata de desplegar el software a producción con solo presionar un botón. Por lo tanto, en todas las prácticas de desarrollo continuo se tratan de evitar tareas manuales propensas a errores. Por lo tanto, se ha cuestionado a los proyectos sobre sus mecanismos de despliegue a producción. Solo 16 equipos (6%) lo están haciendo de forma automática y 37 (15%) presionando un botón. 51 proyectos (20%) tienen que ejecutar un conjunto de scripts manualmente para que se activen y 21 (8%) tienen un proceso completamente manual. También es importante destacar que 130 proyectos no saben cómo se lleva a cabo este proceso. De esos 130 proyectos, 27 de ellos declararon que la implementación es realizada por un equipo diferente (principalmente OPS) y 103 tienen miembros senior específicos del equipo (más expertos) que son los únicos que pueden realizar esta tarea.

Fig. 32. Gestión de activos.

Martin Fowler afirma lo siguiente: "Coloque todo lo que necesita en el sistema de control de versiones y utilícelo para construir todo el sistema con un solo comando" [4]. Esto significa que todos los activos de software deben ser cohesivos, verificables funcionalmente y ser accedidos desde un repositorio de control de versiones, para que la construcción del software se pueda llevar a cabo a través de un solo comando. La Fig. 32 muestra los activos que están almacenados en el repositorio de control de versiones de cada proyecto. El código principal de la aplicación, los scripts de pruebas automatizadas y los archivos de configuración son los activos que más se almacenan en los repositorios de control de versiones. Además, 136 de los proyectos etiquetan todos sus activos, mientras que 21 no. Sin embargo, 98 equipos etiquetan solo algunos de sus activos. De esos 98 proyectos, 47 etiquetan el código base, 32 el código de las pruebas automatizadas, 8 los scripts de despliegue, 13 los archivos de configuración y 24 los scripts de base de datos.

Por otro lado, la etapa de pruebas es una de las fases claves que contribuyen al aseguramiento de la calidad del producto. Aunque se ha creado una sección para problemas y soluciones relacionadas a las pruebas, también se ha incluido algunas preguntas de pruebas relacionadas con el desarrollo continuo en esta encuesta. Por ejemplo, la Fig. 33 muestra los niveles de pruebas que fueron automatizados por los proyectos (Q2.21 en Anexo B) y la Fig. 12 el número de proyectos que crean pruebas automatizadas de sus defectos (Q2.23 en Anexo B), para evitar que vuelvan a aparecer en el futuro.

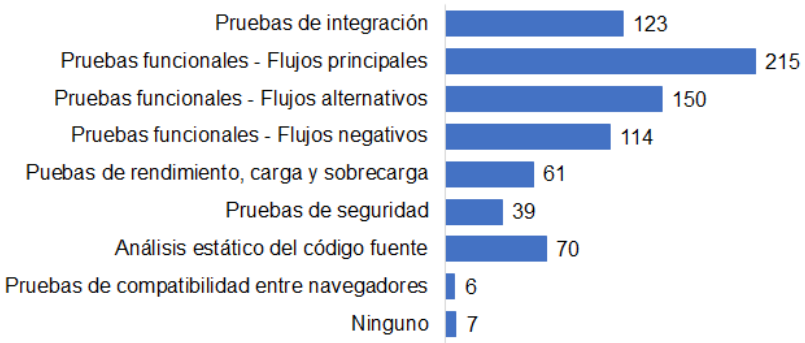


Fig. 33. Niveles de pruebas automatizados.

Aunque la mayoría de los niveles y tipos de prueba están automatizados, hay muchos equipos que no integran esos scripts de prueba automatizados en sus Conductos de Despliegue. La

Tabla 15 muestra cuántos proyectos han automatizado un cierto tipo de nivel de prueba, pero sin incluirlo en la tubería.

Tabla 15. Niveles de pruebas automatizados por proyecto.

Tipo / Nivel	Cantidad de proyectos
Pruebas de integración	46
Pruebas funcionales - Flujos principales	42
Pruebas funcionales - Flujos alternativos	30
Pruebas funcionales - Flujos negativos	31
Pruebas de rendimiento, carga y stress	42
Pruebas de seguridad	35
Análisis estático del código fuente	22

Para descubrir por qué los proyectos no incluyen las pruebas automatizadas mencionadas en la

Tabla 15, se les pidió a los encuestados que den algunas razones (Q2.22 en ANEXO B). Estas se agruparon de la siguiente manera: falta de conocimiento, falta de experiencia, falta de tiempo, falta de herramientas y ambientes inestables. Las mismas se muestran en la Fig. 34.

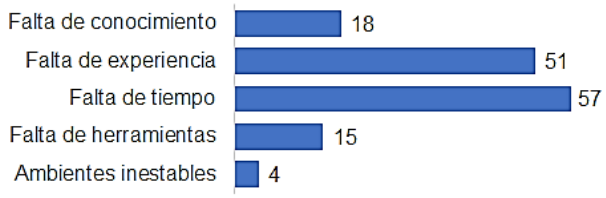


Fig. 34. Causas por las que no se agregan las pruebas automatizadas al DP.

La mayoría de los proyectos justificaron que debido a falta de tiempo y experiencia no han agregado pruebas automatizadas a sus flujos de Integración Continua o a sus Conductos de Despliegue. Sin embargo, también mencionaron que "existe la necesidad de una guía" que contenga los pasos y procedimientos para implementar un proceso de prueba más preciso.

Con respecto a la automatización de casos de pruebas para defectos encontrados, 64 proyectos (25%) declararon que siempre lo hacen, 56 (22%) a veces y 135 (53%) nunca. Estos resultados se pueden ver en la Fig. 35.

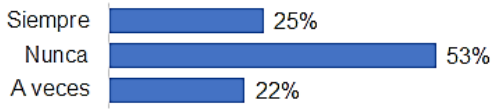


Fig. 35. Pruebas automatizadas para defectos encontrados.

De los 56 equipos que a veces automatizan los casos de pruebas de los defectos encontrados, todos declararon que solo desarrollan scripts para errores de alta gravedad.

Según Duvall [9], realizar inspecciones es otra buena práctica que funciona bien para los equipos que ejecutan Integración Continua en un proyecto. Es importante automatizar esta práctica incorporando una herramienta de análisis estático de código fuente. Al realizar inspecciones, hay muchos atributos que se pueden medir, como código duplicado, cobertura de código, acoplamiento, etc. (ver sección 2.6.1). En la Fig. 36 se presentan los atributos que miden los proyectos. En el Anexo B, la pregunta que reunió estos resultados es Q2.24.

Fig. 36. Atributos que los proyectos miden con las inspecciones de código.

El método seleccionado para gestionar los datos de pruebas es muy importante para lograr un proceso de desarrollo continuo adecuado. En los procesos de pruebas tradicionales, faltaban mecanismos efectivos de generación de datos de prueba [144]. La generación de datos de prueba es uno de los métodos más recomendados en la gestión de datos de pruebas para implementar un proceso de pruebas en EC [7]. La Fig. 37 muestra los diferentes mecanismos de gestión de datos de prueba implementados por los proyectos (Q2.25 en Anexo B).

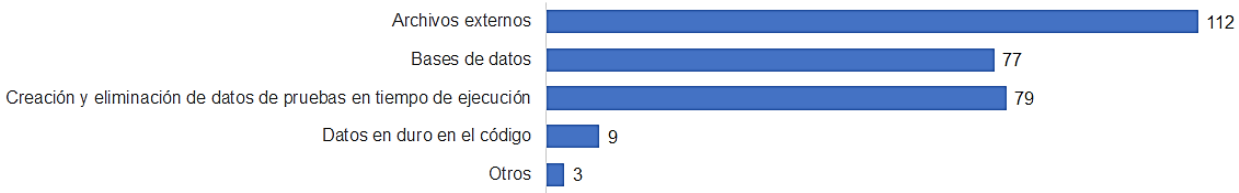


Fig. 37. Gestión de datos de pruebas.

112 proyectos manifestaron que usan fuentes externas como archivos json, hojas de cálculo, archivos csv, archivos xml, entre otros. 79 equipos han implementado un mecanismo de generación de datos de prueba que crea los datos de prueba antes de que se ejecuten las pruebas y luego los elimina una vez que finaliza la ejecución. 77 proyectos almacenan los datos permanentemente en una base de datos y 9 introducen los datos de pruebas en duro en el código (*hardcoding*[36]). Hay otros 3 proyectos que expresaron utilizar un mecanismo híbrido donde las pruebas buscan los datos en una base de datos y si no están allí, los crea y luego los almacena. Una vez que finaliza la ejecución, los datos de prueba no se eliminan.

Finalmente, antes de pasar a la última sección de la encuesta, se buscaba medir el grado de colaboración entre los diferentes roles en el equipo. Mike Cohn [33], Paul Duvall [2], Jez Humble y David Farley [1], afirman que la colaboración entre los miembros de equipo es muy importante para lograr prácticas de desarrollo continuo. Tradicionalmente, los desarrolladores y especialistas en pruebas (o simplemente testers) trabajaban como grupos diferentes, pero en Integración Continua, Entrega Continua o Despliegue Continuo, deben trabajar juntos desde el principio. Los testers deben trabajar con el analista para definir los criterios de aceptación de los requisitos del cliente y deben trabajar con los desarrolladores para ayudarlos a comprender estos requisitos desde el principio [7]. Además, los testers deben trabajar con los encargados de automatizar las pruebas para escribir pruebas de aceptación automatizadas [1]. Por otro lado, los desarrolladores deberían ayudar a los testers a probar la aplicación cuando sea necesario. Las pruebas automatizadas también deben agregarse como parte de la Definición de Hecho[37] y deben desarrollarse antes de que finalice la iteración, para que puedan incluirse en los lotes de regresión que se ejecutarán cuando se incluyan nuevas características. Por lo tanto, la Fig. 38 muestra la colaboración entre estos roles y los proyectos que han incluido pruebas automatizadas en sus documentos de Definición de Hecho. Esto se obtuvo con la ayuda de las preguntas Q2.26, Q2.27 y Q2.28 del Anexo B.

Por un lado, en 138 proyectos (55%) hay desarrolladores que ayudan a los testers a probar la aplicación cuando es necesario y 67 proyectos (25%) no siguen esta práctica de colaboración. También hay 31 proyectos (12%) en los que sus desarrolladores ayudan a los testers solo cuando la función a probar requiere habilidades de codificación. Por otro lado, 142 proyectos (55%) tienen especialistas en pruebas que ayudan a los desarrolladores a comprender los requisitos y 69 (27%) no. 25 equipos tienen testers que ayudan a los desarrolladores solo

[36] Hardcoding es una “mala práctica” de desarrollo de software en la cual se introducen datos directamente en el código fuente del programa en lugar de obtenerlos de una fuente externa o de parámetros recibidos.

[37] La definición de hecho, es un acuerdo común que debe existir entre los miembros de un equipo ágil para considerar una pieza de trabajo como terminada (hecha).

cuando tienen tiempo libre. 19 proyectos (8%) manifestaron que no tienen testers en sus equipos. Con respecto a los proyectos que incluyen las pruebas automatizadas como un elemento en la Definición de Hecho, 73 de ellos (29%) están implementando esta práctica y lográndolo en cada iteración. Además, hay otros 51 proyectos (20%) que incluyen las pruebas automatizadas en la Definición de Hecho, pero la mayoría de las veces no lo pueden cumplir. Finalmente, 131 equipos (51%) no implementan esta práctica.

Fig. 38. Colaboración entre los miembros del equipo.

PROBLEMAS Y SOLUCIONES EN EL PROCESO DE PRUEBAS

Como se mencionó anteriormente, según la literatura hay algunos problemas relacionados con las pruebas en entornos de desarrollo continuo. Sin embargo, hubo soluciones propuestas para enfrentar estos problemas. En esta sección de la encuesta, se buscaba obtener los problemas y soluciones con relación a las pruebas en la industria. Con el fin de obtener la mayor cantidad de detalle posible, la mayoría de las preguntas de esta sección eran preguntas abiertas, de modo que los encuestados pudieron expresar todos los problemas que enfrentan y las soluciones o soluciones parciales que pudieron implementar.

La primera parte de esta sección de la encuesta (Q3.1 en el cuestionario del ANEXO B), fue sobre los problemas relacionados con las pruebas que tienen las empresas actualmente. Los problemas se presentan en la Tabla 16

Como se puede ver en la Tabla 16, aparecieron nuevos problemas que no se detectaron en la literatura. Del mismo modo, hubieron problemas que se encontraron en la literatura y que no han sido mencionado por las empresas.

Tabla 16 y en la Fig. 39.

Como se puede ver en la Tabla 16, aparecieron nuevos problemas que no se detectaron en la literatura. Del mismo modo, hubieron problemas que se encontraron en la literatura y que no han sido mencionado por las empresas.

Tabla 16. Problemas relacionados a las pruebas en la industria.

Problema	**Cantidad de proyectos**
Pruebas no deterministas	224
Automatización de pruebas de interfaz gráfica de usuario	124
Automatización de pruebas web con contenido dinámico	47
Resultados de ejecución de pruebas ambiguos	65
Pruebas en Big Data	18
Pruebas de datos	26
Pruebas en dispositivos móviles	17
Pruebas de servicios web	58
Pruebas que consumen mucho tiempo de ejecución	179
Falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo	112
Ambientes inestables	36
Pruebas automatizadas de requerimientos no funcionales	23
Pruebas de aplicaciones compuestas por servicios en la nube	2

Los nuevos problemas que se encontraron son:

- Falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo.
- Ambientes inestables.

Las pruebas no deterministas fue el problema más reportado. Según los encuestados, hay varias causas de pruebas no deterministas, entre en las que se incluyen fallos por: datos de prueba faltantes o dañados, componentes de GUI faltantes, tiempos de espera agotados debido a problemas del entorno, elementos dinámicos que tardan demasiado en cargar e inconsistencias en el código de las pruebas. La mayoría de los encuestados declararon que las pruebas no deterministas son la razón principal para tener pruebas automatizadas en un flujo de Integración Continua diferente del de la construcción y pruebas unitarias.

Con respecto a las pruebas automatizadas de GUI, el problema más difícil informado es el mantenimiento de los scripts de pruebas debido a cambios en los localizadores de los elementos GUI[38]. Además, las pruebas automatizadas de GUI tardan demasiado en ejecutarse. Del mismo modo, ejecutar pruebas automatizadas en un sitio web que tiene elementos web dinámicos es un desafío para los equipos de desarrollo, ya que una prueba puede fallar porque la página ya ha cargado por completo y el elemento aún no se muestra, debido a llamadas asíncronas. Las pruebas de GUI web no son las únicas que llevan mucho tiempo de ejecución, sino también las pruebas de servicios web y móviles.

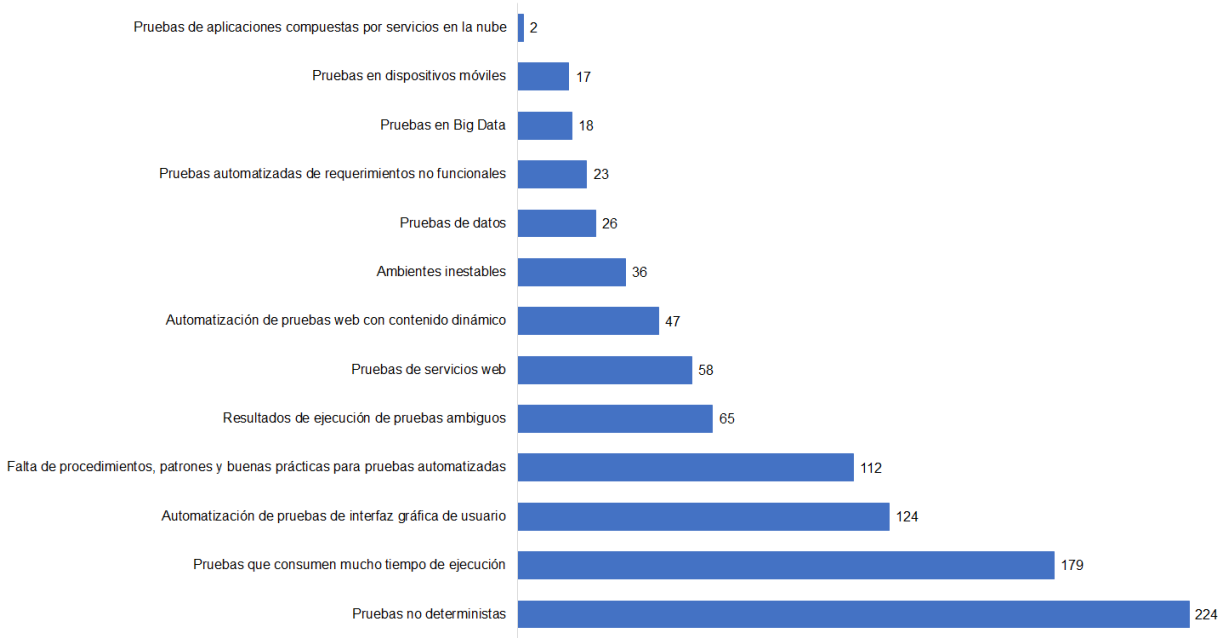


Fig. 39. Problemas relacionados a las pruebas en la industria.

Un grupo de equipos ha declarado que probar los atributos de los datos es otro desafío, especialmente si las pruebas tienen que ser automatizadas. En el mismo contexto, hay muchos

[38] Los localizadores de elementos GUI son sentencias de código en las pruebas que permiten interactuar con componentes en la GUI de una aplicación, como por ejemplo un botón o un formulario.

proyectos que implementan tecnologías de Big Data y la mayoría de ellos ha dicho que hay una falta de documentación en Internet sobre las Pruebas en Big Data, incluida la verificación de diferentes etapas como extracción de datos, procesamiento de datos, transformación de datos y presentación de datos. Algunos otros equipos reportaron problemas con las pruebas móviles y las pruebas de servicios web, todos ellos relacionados con la falta de marcos de trabajo y modelos maduros, incluida la falta de documentación para implementarlos adecuadamente en entornos de desarrollo continuo. Los mismos problemas fueron mencionados por proyectos que desarrollan aplicaciones compuestas por servicios en la nube.

Para los desafíos de pruebas automatizadas no funcionales, 17 proyectos reportaron inconvenientes a la hora de automatizar pruebas de rendimiento, carga y stress que puedan ser luego integradas al DP. Otros 6 proyectos tienen problemas con la falta de información sobre las pruebas de seguridad en entornos de desarrollo continuo. Además, de acuerdo con 65 proyectos, los reportes de pruebas automatizadas son difíciles de comprender (por ejemplo, es difícil entender cuál fue la causa por la que una prueba falló).

Finalmente, algunos proyectos declararon que no existe una guía como "regla general" con buenas prácticas y procesos para lograr una buena cobertura de pruebas automatizadas para diferentes partes de la aplicación. Diferentes equipos han mencionado que hay una falta de documentación formal sobre las estrategias de gestión de datos de pruebas y la gestión del marco de trabajo de automatización de pruebas. Otros proyectos indicaron que, si bien hay muchos tutoriales, capacitaciones y documentación sobre la automatización de pruebas tradicionales, hay falta de información sobre los procesos de automatización de pruebas en entornos de desarrollo continuo. En resumen, existe una carencia de procedimientos, patrones y buenas prácticas para las pruebas automatizadas en entornos de desarrollo continuo.

Afortunadamente, algunas compañías han enfrentado estos y otros problemas y han encontrado soluciones parciales o totales (Q3.2 en Anexo B).

Soluciones para pruebas no deterministas: 17 proyectos han implementado la técnica de generación de datos de prueba como una forma de evitar pruebas no deterministas producidas por datos faltantes o inconsistentes. Otros 4 equipos utilizaron una técnica llamada "estabilización" que consiste en mover todas las pruebas que fallan al azar a una cuarentena y dejar solo las pruebas que no fallan. Sin embargo, eso podría disminuir la cobertura de las pruebas. Del mismo modo, 7 proyectos han reducido el número de pruebas a ejecutar, a un conjunto de pruebas de humo más pequeño y fácil de controlar. Eso también reduce la cantidad de pruebas no deterministas. Hubo 11 equipos que corrigieron las pruebas que estaban fallando debido a problemas relacionados con los entornos de pruebas. Las soluciones que aplicaron

estos equipos se describen en las soluciones de "entornos inestables" a continuación. Finalmente, hubo 1 proyecto que implementó un análisis histórico de fallas con una combinación de aprendizaje automático, que es capaz de detectar si una falla es realmente una falla o no.

Soluciones para automatización de pruebas de GUI: La mayoría de los proyectos que resolvieron parcialmente este problema, manifestaron que la solución estaba en mejorar la colaboración con los desarrolladores. 21 equipos comenzaron a trabajar estrechamente con los desarrolladores, donde los desarrolladores proporcionan localizadores únicos para los componentes de la GUI y los especialistas en pruebas les notifican sobre la posibilidad de romper un elemento en la página antes de comenzar con el desarrollo de uno nuevo. 3 equipos implementaron "pruebas automatizadas de reconocimiento de imágenes" que se trata de identificar y controlar los componentes de la GUI utilizando imágenes. Sin embargo, dos desventajas informadas de esta técnica son el mantenimiento y la necesidad de una pantalla física (como un monitor) conectada al servidor de IC. Hubo otros 9 equipos que simplemente dejaron de automatizar escenarios complejos y los movieron como verificaciones de pruebas manuales. La última solución informada para este problema fue el uso de llamadas de API para condiciones previas que se realizaron mediante pruebas de GUI. Esta solución fue llevada a cabo por 2 proyectos diferentes.

Soluciones para pruebas automatizadas de contenido web dinámico: Solo 11 equipos implementaron una solución para este problema, que consistió en el uso de métodos de espera adecuados (por ejemplo, funciones de *WebDriverWait* de Selenium).

Soluciones para resultados de pruebas ambiguos: 3 proyectos declararon que ahora están registrando las peticiones y respuestas HTTP para los reportes de pruebas de servicios web y otros 2 están agregando metadatos en los informes de pruebas de GUI. Estos metadatos incluyen información del usuario que ejecutó las pruebas, fecha, hora, entorno, etc. Otros 9 equipos agregaron evidencia visual de fallas como capturas de pantalla (6), gifs (2) y videos (1).

Soluciones para pruebas en dispositivos móviles: Uno de los proyectos declaró que, como no tenían las habilidades y los recursos para probar las pruebas en dispositivos móviles de manera adecuada, contrataron a un equipo externo para realizar esta tarea. Otros 4 equipos comenzaron a usar plataformas web en la nube para pruebas móviles y 2 crearon granjas de dispositivos privados compuestas de diferentes dispositivos móviles.

Soluciones para pruebas que consumen mucho tiempo de ejecución: La solución más reportada para este problema fue la paralelización. 18 encuestados han declarado que han implementado la paralelización para enfrentar este problema, pero con ello el número de pruebas no deterministas aumenta de forma directa. Por otro lado, 4 proyectos han reemplazado la

ejecución de scripts de pruebas de GUI en diferentes navegadores con comparación de imágenes. 9 equipos comenzaron a ejecutar solo un conjunto de pruebas de humo, porque el tiempo de prueba no fue suficiente para ejecutar el total de pruebas de una regresión. También hubo 1 proyecto que agregó una opción de "dar comentarios" en el sitio web, en lugar de ejecutar una batería completa de pruebas. De esta manera, el usuario final también brindaba retroalimentación del sitio. 6 equipos dejaron de ejecutar pruebas de GUI y las reemplazaron con pruebas de API. Del mismo modo, otros 2 proyectos reemplazaron su controlador de navegador para las pruebas de GUI con un navegador que se ejecuta en memoria sin GUI. Otros 2 proyectos diferentes implementaron una técnica llamada "Selección de pruebas" que utiliza un mecanismo para detectar los casos de pruebas relacionados con un cambio de código específico y luego ejecuta solo esas pruebas. El mecanismo se basa en los mensajes que se introducen junto con el código en el sistema de control de versiones, que coinciden con las etiquetas que tienen los métodos de prueba. 1 proyecto comenzó a almacenar sus datos de prueba en la base de datos porque eso redujo el tiempo utilizado para extraer y analizar datos de un archivo externo. Finalmente, como una solución para las pruebas manuales que requieren mucho tiempo, 3 equipos han comenzado a automatizar sus escenarios negativos que fueron ejecutados previamente por un equipo de especialistas en pruebas manuales.

Soluciones para ambientes inestables: Como un problema relacionado con pruebas no deterministas, 9 proyectos han estabilizado sus entornos al agregar más recursos de hardware a los servidores. 4 equipos han eliminado la dependencia con los equipos de infraestructura, por lo que son dueños de sus entornos de prueba. Por el contrario, en las diferentes compañías donde otros 6 proyectos tienen este problema, se han creado equipos especiales de infraestructura (la mayoría de ellos equipos OPS).

Soluciones para pruebas automatizadas no funcionales: Uno de los principales problemas en esta categoría es la falta de calidad del código. Por lo tanto, 8 proyectos han agregado herramientas de inspección de código para medir la calidad del código.

3.4.1.3. Discusión de los resultados de la encuesta

En la sección anterior se ha presentado los resultados encontrados en la encuesta. A partir de ahí, para los problemas encontrados en el uso de prácticas de desarrollo continuo y los problemas de prueba reportados, se puede realizar un análisis más profundo y, por lo tanto, se pueden tomar varias conclusiones.

Con respecto a las construcciones y compilaciones privadas (en la computadora del desarrollador), la mayoría de los proyectos están construyendo (90%) y ejecutando las pruebas unitarias (73%) antes de integrar los cambios de código en la rama principal del repositorio de control de versiones. Sin embargo, solo el 33% de los proyectos ejecuta pruebas automatizadas en su estación de trabajo local antes de integrar el código. Según los resultados obtenidos en la sección de problemas reportados, la razón principal es el tiempo: los scripts de pruebas tardan demasiado en ejecutarse. Además, de los 255 encuestados, solo el 24% de ellos realiza varias integraciones de código por día y el 22% de ellos realiza una por día. En las siete mejores prácticas de Integración de Continua [9] se menciona que el código debe ser integrado con frecuencia.

En los flujos de Integración Continua, se pueden encontrar diferentes etapas y niveles de pruebas. Sin embargo, las notificaciones automáticas y las pruebas no funcionales automatizadas son poco consideradas. Además, las pruebas manuales como etapa en el Conducto de Despliegue y las inspecciones de código podrían recibir un poco más de atención. El sistema de notificaciones automático y las pruebas manuales no se han reportado como problemas, pero las pruebas no funcionales automatizadas se informaron como un problema para 23 equipos y también 8 proyectos han tratado de enfrentar este problema mediante la inclusión de herramientas de análisis de código estático.

De los 255 proyectos, 61 tienen dos flujos de Integración Continua, uno para las etapas de construcción, compilación y pruebas unitarias y el otro para las pruebas de aceptación automatizadas. Según los equipos, esto se debe a que las pruebas de aceptación automatizadas producen pruebas no deterministas y las fallas en el Conducto de Despliegue no son confiables. Aparte de eso, solo el 26% de los proyectos ejecuta las pruebas automatizadas automáticamente después de cada integración de código.

Con respecto a las buenas prácticas de Integración Continua, las mejores prácticas implementadas son: tener un entorno similar a producción, realizar pruebas exploratorias para nuevas características, reparar el servidor de Integración Continua inmediatamente cuando se rompe, almacenar diferentes activos en el repositorio de control de versiones. Las prácticas que necesitan más atención son: tener todo el equipo con acceso de lectura y escritura a la base de datos, usar entornos limpios para ejecutar pruebas automatizadas, tener pruebas automatizadas estables en el servidor Integración Continua, tener mecanismos de *rollback*, participar en despliegues a producción. Es evidente la necesidad de DevOps [145] como cultura. La mayoría de los equipos han usado incorrectamente el término DevOps para las personas que trabajan solo en infraestructura.

Las pruebas que más se automatizan son las pruebas funcionales positivas (camino feliz). La mayoría de los proyectos lo hacen. En segundo lugar, los equipos están automatizando las pruebas funcionales de rutas alternativas y negativas y también las pruebas de integración. Las pruebas poco automatizadas son las no funcionales como las de rendimiento, carga, estrés, seguridad, capacidad de mantenimiento, etc. Algunos proyectos (6) tienen pruebas de navegación cruzada automáticas para verificar la compatibilidad de un sitio web entre diferentes navegadores. Finalmente, 7 proyectos no han automatizado sus pruebas (solo pruebas unitarias).

No todos los proyectos han agregado las pruebas automatizadas a sus Conductos de Despliegue y más de la mitad de los proyectos no automatizan las pruebas para defectos. Las causas principales son la falta de tiempo y la falta de experiencia, pero también dijeron que "existe la necesidad de una guía" que contenga los pasos y procedimientos para implementar un proceso de prueba preciso.

Con respecto a las inspecciones de código, los equipos que lo implementan están midiendo los diferentes atributos correctamente. Sin embargo, solo 103 proyectos de los 255 están realizando análisis de código estático.

En cuanto a los resultados para la gestión de datos de prueba, solo un 30% de los encuestados declararon que están utilizando una técnica automatizada de generación de datos de pruebas. Otros 3 equipos están utilizando esta técnica, pero solo para crear datos, no para eliminarlos después de su uso. Esto podría significar la falta de conocimiento del equipo de esta técnica como un método eficaz para reducir las pruebas no deterministas relacionadas con datos de pruebas faltantes o incorrectos.

Con respecto a la colaboración en equipo, a pesar de que los porcentajes de colaboración en equipo están por encima del 50%, la colaboración debe ocurrir en todos los proyectos. Por otro lado, el número de proyectos que incluyen pruebas automatizadas en sus Definiciones de Hecho es muy bajo y el 41% de ellos no pueden completar esta tarea al final de la iteración.

Por último, en los resultados relacionados con los problemas de prueba reportados, los más reportados fueron pruebas no deterministas, pruebas que requieren mucho tiempo, pruebas automatizadas de GUI y falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo.

3.5. Relevamiento de los problemas más importantes en Pruebas Continuas

Partiendo de los problemas detectados tanto por la RSL, como por la encuesta, es posible determinar cuáles son los problemas en Pruebas Continuas más importantes. En un estudio llevado a cabo recientemente se realizó un relevamiento de estos problemas mediante un análisis comparativo entre la RSL y la encuesta [146]. El mismo se presenta en detalle en el ANEXO C. La

Tabla 17 representa la relación entre los problemas encontrados en la literatura académica y los encontrados en la encuesta.

Tabla 17. Relación entre los problemas encontrados en la literatura y la industria con su severidad.

Problema	Literatura	Industria	Severidad
Pruebas no deterministas.	Presente	Presente	Muy alta
Pruebas que consumen mucho tiempo de ejecución.	Presente	Presente	Alta
Pruebas automatizadas de interfaz gráfica de usuario.	Presente	Presente	Alta
Pruebas en Big Data.	Presente	Presente	Alta
Falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo.	Faltante	Presente	Media
Pruebas en dispositivos móviles.	Presente	Presente	Media
Resultados de ejecución de pruebas ambiguos.	Presente	Presente	Baja
Pruebas automatizadas web con contenido dinámico.	Presente	Presente	Baja
Pruebas de servicios web.	Presente	Presente	Baja
Pruebas automatizadas de requerimientos no funcionales.	Presente	Presente	Muy baja
Pruebas de datos.	Presente	Presente	Muy baja
Ambientes inestables.	Faltante	Presente	Muy baja
Pruebas de aplicaciones con servicios en la nube.	Presente	Presente	Muy baja
Pruebas como un servicio.	Presente	Faltante	Muy baja

4. MODELO DE MEJORA PARA PRUEBAS CONTINUAS

Como se ha visto en la sección 3.5, los problemas de pruebas más críticos en entornos continuos son:

1. Pruebas no deterministas.
2. Pruebas que consumen mucho tiempo de ejecución.
3. Pruebas automatizadas de interfaz gráfica de usuario.
4. Pruebas en Big Data.

También existen problemas de severidad media, los cuales son:

5. Pruebas en dispositivos móviles.
6. Falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo.

En primer lugar, las pruebas no deterministas se presentan en todo tipo de plataformas: web, escritorio, dispositivos móviles, etc. Según una RSL de problemas en Entrega Continua [12] y un análisis empírico relacionado a pruebas no deterministas [147], las causas más comunes son: esperas agotadas, servidor no disponible, ambiente de pruebas inestable, falta de calidad en el código de las pruebas automatizadas y pruebas de GUI. Éstas últimas también representan una problemática crítica como tal y están vinculadas a las pruebas de plataformas que tienen una interfaz visual sea web, móvil o de escritorio. Esto se debe, principalmente, a los cambios en los elementos internos de la interfaz.

Por otro lado, el tiempo es un factor clave en los entornos continuos y se evidencia la necesidad de un proceso de organización, selección y optimización del tiempo de ejecución de las pruebas. Las pruebas en dispositivos móviles también presentan inconvenientes de severidad media en los entornos de desarrollo continuo. Como se mencionó en las secciones 3.2.2 y 3.4, si bien existen herramientas para automatizar las pruebas en este tipo de plataformas, hay una carencia buenas prácticas acerca de infraestructura y arquitectura. Asimismo, las pruebas en Big Data demuestran tener una severidad alta según los estudios realizados.

Finalmente, de forma general, se menciona una falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo que puedan ayudar a solucionar los problemas antes mencionados. Por ejemplo, para las esperas agotadas, existen métodos de espera [7] y para los servidores no disponibles o ambientes inestables, existe el método de ejecución de pruebas de salud [20]. Para la falta de calidad en el código de las pruebas automatizadas, se pueden seguir patrones de diseño [148] y revisiones de código. Del mismo

modo, existen soluciones que pueden ser utilizadas para paliar los problemas con las pruebas de GUI. Todos estos inconvenientes, son los que producen las pruebas no deterministas.

Además, en la literatura académica y en ejemplos de la industria, se han presentado una gran cantidad de propuestas para hacer frente a una parte de cada problema mencionado en los ambientes de desarrollo continuo. De esta manera, reuniendo y analizando todas estas propuestas, es posible la construcción de un conjunto de buenas prácticas estructuradas para adoptar las Pruebas Continuas en una organización, de forma gradual. Esta receta se propone a continuación como un modelo formado por un conjunto de buenas prácticas agrupadas para aplicar, siguiendo niveles de adopción, denominados niveles de mejoras. La configuración en niveles sigue la estructura utilizada en otros modelos de mejora de procesos, como los vistos en la sección 2.7 y cuyo objetivo es asegurar la implantación sostenida a lo largo del tiempo favoreciendo la institucionalización de las mejoras. A continuación, se presenta este modelo propuesto.

4.1. Definición del Modelo

El siguiente modelo es una propuesta para facilitar la planificación, implementación, mantenimiento y mejora continua, de la etapa de pruebas en entornos de desarrollo continuo. El modelo propone cuatro niveles de mejoras, llamados **Niveles de Mejora de Pruebas Continuas** (CTIL[39]). Por esta razón el modelo se denomina **Modelo de Mejora para Pruebas Continuas** (CTIM[40]).

Asimismo, el modelo contempla un conjunto de etapas de verificación y validación (V&V) que abarcan diferentes tipos de pruebas y procedimientos que han sido definidos y propuestos por diferentes autores tanto en la industria como en la literatura académica. Estas etapas V&V permiten formar el **Conducto de Pruebas Continuas** (CTP[41]), la cual se muestra en la Fig. 40, análogo al concepto de conducto utilizado en el despliegue o la Entrega Continua, descrito en la sección 2.5.2.

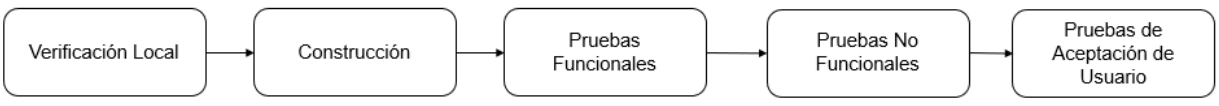


Fig. 40. Conducto de Pruebas Continuas.

[39] *Continuous Testing Improvement Level*
[40] *Continuous Testing Improvement Model*
[41] *Continuous Testing Pipeline*

4.1.1. Etapas de Verificación y Validación

Como se muestra en la Fig. 40, las etapas de V&V son: verificación local, construcción, pruebas funcionales, pruebas no funcionales, pruebas de aceptación de usuario. Estas etapas son seleccionadas partiendo de un conjunto de buenas prácticas y patrones propuestos por los autores de Integración y Entrega Continua [4], [7], [9], [10], las cuales han sido mencionadas a lo largo de la sección 2.

4.1.1.1. Etapa de verificación local

La etapa de verificación local ocurre antes que el código sea integrado en el tronco principal, del repositorio de control de versiones. En la introducción del libro de Integración Continua [9], Paul Duvall expresa lo siguiente: *"Muchos equipos encuentran a la Integración Continua como una solución que reduce significativamente los problemas de integración y permite que un equipo desarrolle software cohesivo más rápidamente. En mi experiencia, esto significa cosas diferentes, pero, en primer lugar, que todos los desarrolladores ejecuten construcciones y compilaciones privadas en sus propias estaciones de trabajo antes de enviar su código al repositorio de control de versiones para garantizar que sus cambios no rompan el servidor de integración".*

Una construcción y compilación privada del sistema se realiza para disminuir las posibilidades de que los cambios recientes del desarrollador corrompan el servidor de Integración Continua y hagan que la construcción no termine de manera exitosa. Es por ello que la ejecución de la construcción del código de manera privada es una de las buenas prácticas de Integración Continua [9]. También es importante que el código fuente local sea actualizado de forma constante para tener la misma versión que se encuentra en el repositorio de control de versiones.

Sin embargo, si bien la construcción y la compilación del código son tareas que disminuyen el riesgo de corromper el servidor de Integración Continua en la primera etapa, no lo disminuye para etapas posteriores.

4.1.1.2. Etapa de construcción

La etapa de construcción es la primera etapa que se ejecuta en el servidor de Integración Continua. Su objetivo es afirmar que el sistema funciona en un nivel técnico. Los pasos son los siguientes: realiza una compilación, ejecuta un conjunto de pruebas unitarias y lleva a cabo el

análisis de código de manera estática. También es responsabilidad de esta etapa generar los archivos binarios o instaladores del software que conformarán la versión candidata[42].

Esta etapa inicia su ejecución cuando los desarrolladores integran los cambios en la rama principal del sistema de control de versiones. En este punto, el servidor de Integración Continua detecta los cambios introducidos y responde con la ejecución de la etapa de construcción. Si la construcción es exitosa y las pruebas unitarias no fallan, el código se convierte en ejecutable y es almacenado en un repositorio de artefactos. Los servidores de Integración Continua modernos proporcionan una infraestructura para almacenar artefactos como estos y hacerlos fácilmente accesibles tanto para los usuarios como para las etapas posteriores del flujo. Alternativamente, del mismo modo que existen herramientas de repositorio de versiones, existen muchas herramientas para administrar artefactos. Los servidores modernos permiten ejecutar las tareas de esta etapa en paralelo reduciendo los tiempos de ejecución del conducto de despliegue.

4.1.1.3. Etapa de pruebas funcionales

Las pruebas funcionales son el mecanismo principal para verificar el cumplimiento de los requerimientos funcionales del sistema. Como se menciona anteriormente, dependiendo del sistema, las pruebas funcionales se realizan sobre una GUI o sobre una API. La GUI puede ser un sitio web que se accede mediante un navegador, un programa de escritorio que se instala en el ordenador del usuario o una aplicación ejecutada desde un dispositivo móvil. La API puede ser un servicio web, una interfaz de líneas de comandos, o cualquier otro tipo de interfaz de sistema que no sea gráfica.

En los entornos continuos esta etapa debe ser totalmente automatizada y ejecutada por el servidor de Integración Continua, luego de la etapa de construcción. En la versión inicial del modelo, esta etapa se llamaba pruebas de aceptación, pero fue modificada por sugerencia de los expertos (sección 6.1.4), ya que representa mejor las diferentes prácticas que la componen.

A medida que se vayan completando los niveles CTIL en esta etapa, se tendrá mayor cobertura de pruebas funcionales automatizadas, las cuales serán ejecutadas con el mayor grado de eficiencia y optimización. Mayor cobertura no solo significa la mayor cantidad de escenarios positivos, sino también alternativos y negativos, abarcando tanto la verificación de la GUI como las API y los flujos de datos.

[42] Una versión candidata a definitiva, candidata a versión final o candidata para el lanzamiento (del inglés release candidate), comprende un producto preparado para publicarse como versión definitiva a menos que aparezcan errores que lo impidan. https://es.wikipedia.org/wiki/Ciclo_de_vida_del_lanzamiento_de_software#Versi%C3%B3n_candidata_a_definitiva_(RC)

Como se indicó en la sección 2.6.4, una de las principales ventajas de ejecutar pruebas funcionales automatizadas es asegurarnos de que un cambio de código no ha afectado negativamente a las funciones existentes.

4.1.1.4. Etapa de pruebas no funcionales

La mayoría de los sistemas tienen gran cantidad de requerimientos no funcionales [7]. Por ejemplo, requisitos sobre rendimiento y seguridad, o los acuerdos de nivel de servicio que deben cumplir.

Como se menciona en la sección 2.6.4.2, esta etapa abarca la ejecución de pruebas para cuatro tipos de requerimientos no funcionales: rendimiento, usabilidad, accesibilidad y seguridad. Asimismo, en las primeras etapas de V&V, se realiza una verificación de mantenibilidad, mediante la ejecución de análisis estático del código fuente. El resto de las pruebas no funcionales se ejecutan en esta etapa.

4.1.1.5. Etapa de pruebas de aceptación de usuario

Uno de los principios de Entrega Continua es automatizar todo lo que sea posible [7]. Sin embargo, algunas actividades no se pueden automatizar: las pruebas exploratorias requieren de expertos, las demostraciones del software al cliente no pueden ser realizadas por computadoras y las aprobaciones requieren intervención humana. Estas tareas mencionadas son las más importantes, ya que conforman la aceptación final de los cambios introducidos, por diferentes partes como se ha indicado en la sección 2.6.5. La etapa de pruebas de aceptación de usuario comprende estas actividades.

En una primera etapa se incluyen estrategias de pruebas manuales y se aumenta la frecuencia de las demostraciones del software. Luego, esta estrategia pasa a ser registrada para su posterior automatización en otras etapas previas.

4.2. Niveles de Mejoras

Cada nivel de CTIM indica una jerarquía de mejora y un camino evolutivo para evaluar el progreso del proceso de pruebas de un proyecto de desarrollo en entorno continuo. Cada nivel tiene un conjunto de actividades relacionadas a tipos de pruebas o tareas técnicas que conforman una etapa de V&V y que en su conjunto se pueden considerar un paso general de mejora. De

este modo, un proyecto o equipo puede alcanzar un cierto nivel de mejora para una determinada etapa de V&V, denominándose mejora parcial. Una vez que se cumplan todas las actividades propuestas para un CTIL, la mejora será total en ese nivel. Los esfuerzos de mejora del proceso de pruebas deben centrarse siempre en las necesidades de la organización en el contexto de su entorno empresarial, los servicios que presta o sus clientes.

Sin embargo, de forma similar a otros modelos [105], cada CTIL forma una base necesaria para el siguiente nivel. Estos niveles de mejora propuestos para las Pruebas Continuas se muestran en la Fig. 41.

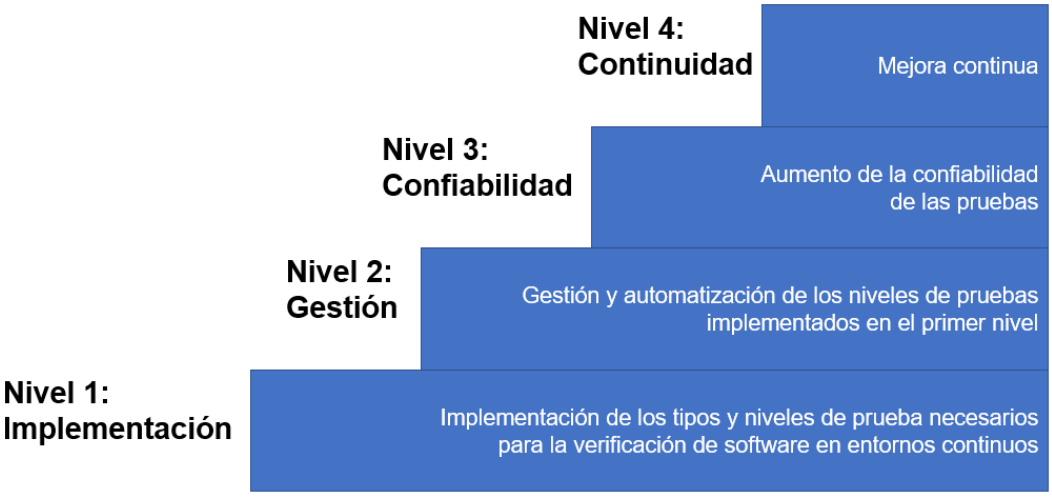


Fig. 41. Niveles de mejora para Pruebas Continuas.

La selección de estos niveles de mejora surge del análisis de los niveles de capacidad y madurez propuestos por CMMI [107] y TMMI [105] de acuerdo a lo escrito en la sección 2.7. La relación entre estos niveles se presenta en la Tabla 18.

Tabla 18. Relación entre los niveles de capacidad (CMMI), madurez (CMMI y TMMI) y mejora (CTIM).

Nivel	Niveles de capacidad (CMMI)	Niveles de madurez (CMMI/TMMI)	Niveles de mejora (CTIM)
Nivel 0	Incompleto		
Nivel 1	Realizado	Inicial	Implementado
Nivel 2	Gestionado	Gestionado	Gestionado
Nivel 3	Definido	Definido	Confiable
Nivel 4		Gestionado cuantitativamente	Continuo
Nivel 5		En optimización	

En el nivel de mejora 1, **se implementa** la automatización de pruebas manuales y se agregan más tipos de pruebas que son necesarias en los entornos continuos. La automatización es el requisito para lograr Pruebas Continuas.

En el nivel de mejora 2, el proceso de pruebas **se gestiona** y planifica y por ende las pruebas son estructuradas y organizadas de acuerdo a diferentes criterios como por ejemplo la severidad o la prioridad de una funcionalidad. No se pueden someter a las pruebas a un proceso de mejora y optimización, si no se encuentran adecuadamente estructuradas [11].

En el nivel de mejora 3, se definen estándares y patrones para el proceso de pruebas y, en consecuencia, la etapa de ejecución de pruebas genera resultados **confiables**. La confiabilidad es otro requisito fundamental de las Pruebas Continuas, ya que el lanzamiento del software a producción con solo presionar un botón requiere de pruebas confiables que aseguren la no introducción de errores tal y como se describió en la sección 0. La confiabilidad no solo significa pruebas deterministas, sino también mayor cobertura en su ejecución [15].

En el nivel de mejora 4, el proceso de pruebas comprende la construcción y ejecución de pruebas automatizadas (nivel 1), organizadas (nivel 2), confiables (nivel 3) y eficientes, es decir, Pruebas Continuas. La eficiencia viene dada con la implementación de las soluciones y herramientas propuestas por diferentes autores y empresas para hacer frente a los problemas más críticos reportados en Pruebas Continuas (sección 3.5).

Durante la implementación de estos niveles, pueden presentarse muchos obstáculos. Para evitarlos, es necesario tener presente que un cambio organizacional exitoso no puede ser completamente de arriba hacia abajo o de abajo hacia arriba [3].

Por un lado, si el cambio se hace desde arriba hacia abajo, por más exitoso y respetado que sea el líder o el gerente de proyecto, es imposible brindar correctamente su visión, comunicarla adecuadamente a todas las personas en el proyecto u organización, eliminar obstáculos claves y sobre todo, controlar exhaustivamente todos los cambios que se estén realizando [3], [149].

Por otro lado, en un cambio de abajo hacia arriba, cuando un equipo o algunos miembros del equipo descubren que se necesita un cambio, comienzan a implementarlo. Luego, al encontrarse sin el apoyo de la parte superior de la organización, existirá una resistencia que no se puede superar desde abajo [3]. Esto generalmente ocurre tan pronto como el nuevo proceso comienza a afectar a las áreas externas del equipo original, es decir, cuando se inicia un cambio en la manera en que realizan sus actividades actuales.

Por esta razón, es necesario combinar ambos modos [3], [150]: supervisores que ayuden a superar las barreras de la resistencia y miembros de equipo que sean capaces de entender por

qué el nuevo proceso mejora el anterior, encontrar las dificultades y hasta proponer otras mejoras. Además, un experto debe acompañar la implementación de las prácticas que componen el modelo CTIM a lo largo de todo el proceso.

Como se muestra en el análisis de problemas de Pruebas Continuas (Fig. 26), los mismos se agrupan en **pruebas que consumen mucho tiempo de ejecución, pruebas automatizadas no deterministas y** un tercer grupo de **poca cobertura de pruebas en Entrega Continua**, representando a los problemas relacionados con dificultades en la implementación de ciertos tipos de pruebas, debido a la carencia de herramientas, modelos o conjunto de buenas prácticas para pruebas automatizadas.

La implementación de cada nivel CTIL del modelo implica una mejora gradual en cada uno de esos grupos de problema. Este nivel de mejora gradual será representado como un gráfico de área, resultante de medir el impacto directo e indirecto de cada buena práctica propuesta por el CTIL sobre el problema de Pruebas Continuas.

4.2.1. Nivel de Mejora 1

El nivel de mejora 1, llamado **implementación**, tiene como objetivo la puesta en marcha de los diferentes tipos y niveles de pruebas necesarios en el ciclo de vida del proyecto. Los tipos y niveles de pruebas seleccionados surgen de una recopilación de artículos relacionados a las pruebas en los entornos continuos [3], [4], [7], [10], [28], [78], [79], [83], [151]. Adicionalmente, no se han reportado otro tipo de pruebas utilizadas en estudios empíricos [11].

En cada etapa de V&V, se propone la implementación de ciertos tipos de pruebas, como también la automatización de pruebas que son realizadas de forma manual, como las ejecuciones de pruebas funcionales de GUI que son llevadas a cabo generalmente por especialistas en pruebas o por los mismos desarrolladores. Por esta razón, este es un nivel de implementación donde aún no se proponen buenas prácticas.

El conducto de despliegue para Pruebas Continuas de este nivel puede verse en la Fig. 42, donde se aprecia cual es el objetivo final antes de continuar con el siguiente nivel. Como se mencionó anteriormente, este es uno de los niveles en donde se necesita el principal apoyo de un supervisor ya que muchas partes pueden mostrar resistencia al cambio [3].

4.2.1.1. Etapa de verificación local

Como se menciona anteriormente, la etapa de verificación local ocurre antes que el código sea integrado en la línea troncal o rama principal repositorio de control de versiones. En la primera versión del modelo, esta etapa se llamaba “pre-compilación”, pero debido a las recomendaciones de los expertos que lo validaron en el apartado 6.1, fue renombrada a “verificación local”. Las prácticas que componen esta etapa en este nivel inicial se presentan a continuación.

PRÁCTICA 1.1.1: PRUEBAS UNITARIAS AUTOMATIZADAS

Las pruebas unitarias deben automatizarse para su posterior ejecución junto con la construcción y la compilación del código antes de integrar el código con el repositorio de control de versiones. En el artículo de Rodríguez et al. [89] se listan muchas de las opciones que existen en la actualidad para implementar pruebas unitarias. La ejecución de este tipo de pruebas puede realizarse utilizando un comando por consola, como también mediante un plugin en el entorno de desarrollo integrado. Las pruebas son ejecutadas de forma secuencial, una después de la otra y en caso de que una falle, no se interrumpe el resto de la ejecución.

PRÁCTICA 1.1.2: REVISIONES DE CÓDIGO

Complementariamente a la automatización de pruebas unitarias, en este nivel se aplican las revisiones de código para validar el diseño y la implementación de la funcionalidad que se está desarrollando. Esto ayuda a los desarrolladores a mantener la coherencia entre los estilos de diseño y estándares de implementación que se siguen en el proyecto.

4.2.1.2. Etapa de construcción

Es imposible tener un CTP sin un servidor de Integración Continua (ver sección 2.3.1) y la primera tarea automatizada de este servidor, es la construcción del código [7].

PRÁCTICA 1.2.1: CONSTRUCCIÓN AUTOMÁTICA

Lo que se busca es la creación y configuración de una tarea automatizada de construcción. Su objetivo en el Conducto de Pruebas Continuas es eliminar las construcciones de código que no son aptas para producción e interrumpir el flujo lo antes posible.

Es importante configurar a la tarea automatizada un sistema de retroalimentación que notifique al desarrollador y otras partes interesadas si la construcción ha fallado o ha sido exitosa.

PRÁCTICA 1.2.2: EJECUCIÓN AUTOMÁTICA DE PRUEBAS UNITARIAS

También es responsabilidad de esta etapa la ejecución de las pruebas unitarias que fueron agregadas en la etapa anterior. Por esta razón, en esta etapa se debe ejecutar el mismo script o comandos que se ejecutan en la etapa de verificación local, tanto para la construcción como para la ejecución de pruebas unitarias.

4.2.1.3. Etapa de pruebas funcionales

Cualquier tipo de prueba con sus escenarios que se realice para verificar el funcionamiento de la aplicación, pertenece a esta etapa. Dependiendo del tiempo del que se disponga en el proyecto y la cantidad de recursos dedicados al proceso de pruebas, este registro puede ser simplemente un listado de todos los escenarios que se ejecutan para verificar las funcionalidades del sistema, como también un conjunto de casos de pruebas bien documentados.

Es importante señalar que, en los entornos continuos, la etapa de pruebas funcionales abarca únicamente las regresiones, donde solo se prueban funcionalidades previamente desarrolladas. La verificación de la funcionalidad que se está desarrollando y cuyos cambios son introducidos en el flujo de Integración Continua, se realiza mediante pruebas de aceptación, que pertenecen a otra etapa de Pruebas Continuas.

PRÁCTICA 1.3.1: PRUEBAS FUNCIONALES AUTOMATIZADAS

Por lo general, al inicio del cualquier proyecto de desarrollo de software, la ejecución de estos escenarios se realiza de forma manual. Para este nivel CTIL, esta actividad debe comenzar a automatizarse, mediante la incorporación de herramientas de automatización de pruebas funcionales. En la actualidad, existen muchas herramientas de automatización de acuerdo con la necesidad y tecnología de cada proyecto[43].

El marco de trabajo para la construcción de las pruebas funcionales debe ser parte del proyecto donde se encuentra el código base de la aplicación [20]. En lo que respecta al lenguaje

[43] https://testguild.com/automation-testing-tools/

de programación algunos autores recomiendan utilizar el mismo lenguaje que se utiliza en el desarrollo de la lógica de negocio para las pruebas de API y el que se utiliza en la interfaz para las pruebas de GUI. Sin embargo, esto no siempre es posible en los equipos de especialistas en pruebas automatizadas que trabajan separados de los desarrolladores.

PRÁCTICA 1.3.2: COBERTURA DE PRUEBAS FUNCIONALES

El objetivo de esta etapa para este nivel es comenzar la automatización, pero no hacerlo con todos los escenarios que se tengan registrados en el proyecto. Además, es importante implementar algún sistema de etiquetado para los escenarios que ya han sido automatizados para medir el progreso. La cobertura puede obtenerse calculando el total de pruebas automatizadas sobre el total de pruebas que pueden ser automatizadas. Según los expertos que validaron el modelo, las pruebas que no se pueden automatizar también tienen que ser identificadas y eliminadas del cálculo de la cobertura.

4.2.1.4. Etapa de pruebas no funcionales

Los requerimientos no funcionales (RNF) son importantes porque presentan un riesgo de entrega significativo para los proyectos de software. Incluso cuando se tiene en claro cuáles son los RNF del proyecto, es muy difícil llegar al punto óptimo y suficiente para garantizar que se cumplan [7].

PRÁCTICA 1.4.1: MEDICIÓN DE TIEMPOS DE RESPUESTA

Como se ha mencionado en la sección 2.6.4, muchos sistemas fallan porque no pudieron hacer frente a la carga que se les aplicaba, no eran seguros, funcionaban muy lentamente o, quizás lo más común de todo, se volvieron imposibles de mantener debido a la mala calidad del código. En el otro extremo, algunos proyectos fracasan porque se preocupan tanto por los RNF que el proceso de desarrollo es demasiado lento o el sistema se vuelve tan complejo y sobre diseñado que nadie puede descubrir cómo desarrollarse de manera eficiente o apropiada.

En este nivel CTIL, el objetivo de esta etapa es comenzar a tener en cuenta la verificación de este tipo de requerimientos, mediante la incorporación de pruebas de rendimiento, carga o sobrecarga. En la actualidad, existen una gran cantidad de herramientas gratuitas que permiten

realizar pruebas de rendimiento[44]. El primer paso en esta etapa es comenzar a medir el tiempo de respuesta del sistema ante una determinada carga, en un ambiente de prueba y repetir este proceso con cada construcción del código (utilizando la misma carga y ambiente de prueba). De este modo, será posible la detección de anomalías en una construcción de código cuando los tiempos de respuesta estén por encima de la medida de crecimiento obtenidos en las construcciones previas.

4.2.1.5. Etapa de pruebas de aceptación de usuario

Antes del despliegue a producción, el software debe pasar por una etapa de pruebas manuales y una demostración al cliente y/o a un grupo de usuarios [7]. Esto normalmente se realiza desde un entorno de pruebas de aceptación. En esta parte del proceso, el cliente o dueño del producto puede decidir si faltan características. También se pueden encontrar errores que requieren reparación y pruebas automatizadas que deban crearse para aumentar la cobertura.

PRÁCTICA 1.5.1: PRUEBAS DE COMPATIBILIDAD WEB

Las pruebas manuales deben ser llevadas a cabo por un equipo de especialistas expertos en pruebas, con criterio analítico [79]. Esta actividad comprende las denominadas pruebas de aceptación, donde los especialistas verifican que los requerimientos o criterios de aceptación se cumplan, basándose en los casos de pruebas creados durante las etapas de análisis y desarrollo Estas pruebas manuales deben ser ejecutadas en los navegadores y dispositivos más utilizados, para evitar los errores de compatibilidad.

Durante la fase de validación de este modelo (sección 6.2.3), los expertos recomendaron el uso de una herramienta de analítica web, para obtener métricas de los navegadores y dispositivos móviles más utilizados, como por ejemplo Google Analytics[45].

También es importante contar con una lista de verificación durante el ciclo de pruebas para que los especialistas en pruebas manuales contemplen todos los navegadores y dispositivos donde las mismas deben llevarse a cabo (6.2.3.4).

[44] https://www.blazemeter.com/blog/open-source-load-testing-tools-which-one-should-you-use/
[45] https://analytics.google.com/analytics/web/

PRÁCTICA 1.5.2: DEMOSTRACIONES A INTERESADOS EN EL SISTEMA

El software se considera realmente terminado cuando el cliente lo acepta. Tener demostraciones regulares donde esto sucede es la única forma confiable de rastrear el progreso. Los equipos ágiles realizan presentaciones al cliente al final de cada iteración para demostrar la nueva funcionalidad que han entregado (sección 2.2). Sin embargo, si esta tarea se realizara con mayor frecuencia durante el desarrollo, se puede garantizar que cualquier malentendido o problema de especificación se detecte lo antes posible [7].

Durante las demostraciones, por lo general el cliente y los dueños del producto tienen muchas sugerencias para agregar o modificaciones. En este punto, el cliente y el equipo del tienen que decidir cuánto quieren cambiar el plan del proyecto para incorporar estas sugerencias. Cualquiera sea el resultado, es mucho mejor recibir comentarios lo antes posible.

4.2.1.6. Análisis del nivel de mejora 1 y los problemas en Pruebas Continuas

El nivel CTIL 1 propone la utilización de herramientas y marcos de trabajo como el primer paso en la adopción de Pruebas Continuas.

Implica la implementación inicial de cada una de las etapas del CTP y por ello el nivel se denomina de **Implementación**.

El Conducto de Pruebas Continuas para este nivel es presentado en la Fig. 42 y el grado en el que soluciona los problemas principales de Pruebas Continuas se muestran en la Fig. 43.

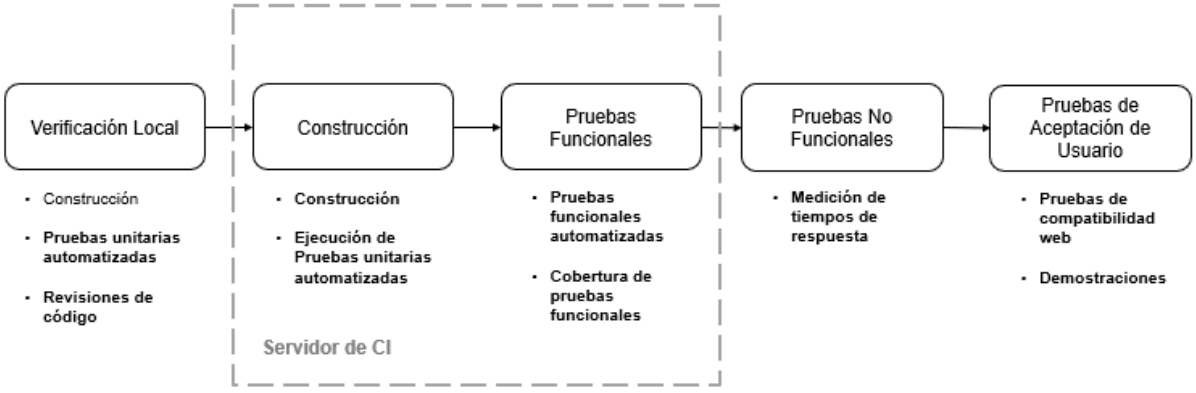


Fig. 42. Conducto de Pruebas Continuas en el nivel CTIL 1.

Con el nivel 1, el problema de poca cobertura de pruebas en Entregas Continuas se soluciona en un 36% (9 problemas de 25). Los tiempos de ejecución elevados se solucionan en un 11,11% (2 problemas de 18) al implementar marcos de trabajo de automatización. Finalmente, no se solucionan los problemas de pruebas no deterministas (0%).

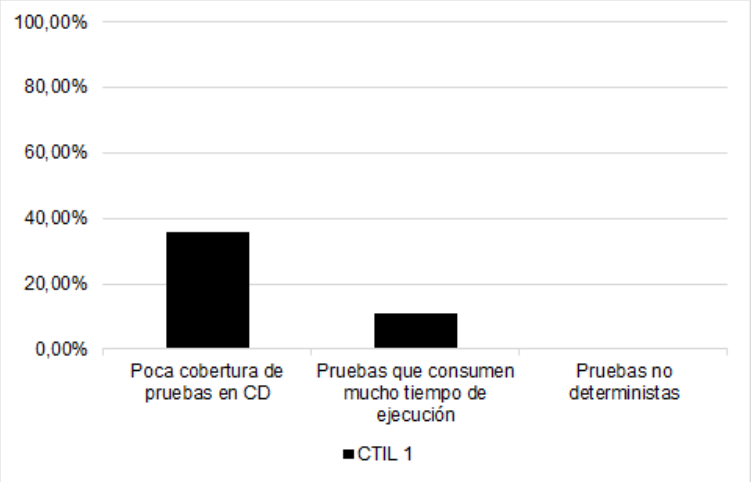


Fig. 43. Grado de resolución de los problemas de Pruebas Continuas al implementar el nivel 1 del modelo.

4.2.2. Nivel de Mejora 2

En el nivel de madurez 2 de CMMI [107] y TMMI [105], los procesos se planifican y se ejecutan de acuerdo con las políticas de la organización. De forma similar, en este nivel CTIL las pruebas también son gestionadas. De acuerdo a la etapa de V&V, las pruebas son agrupadas y organizadas según diferentes criterios: modulo, funcionalidad, sección del sistema y prioridad de ejecución. Esto permite facilidad de ejecución de pruebas en paralelo en otros niveles, como así también la ejecución de pruebas específicas para ahorrar tiempo.

Finalmente, el marco de trabajo de pruebas funcionales y las herramientas también son estructuradas para aumentar su mantenibilidad, escalabilidad y accesibilidad.

4.2.2.1. Etapa de Verificación local

Para este nivel, además de las actividades del CTIL 1, se suman la categorización o agrupamiento de pruebas unitarias utilizando algún criterio en común (módulo, funcionalidad, sección del sistema, etc.). Por ejemplo, para un sistema de ventas de productos que presenta los

módulos de alta de productos, consulta de stock, facturación, gestión de ventas e impresión de comprobantes, las pruebas unitarias se pueden agrupar según esos módulos. La mayoría de los marcos de trabajo de pruebas unitarias que existen en la actualidad tienen la funcionalidad de agrupamiento de pruebas [11].

Adicionalmente, se definen umbrales de cobertura de pruebas unitarias y se implementan las herramientas para la verificación de estos. La cobertura del código es una métrica útil para verificar qué parte del código está bajo el alcance de las pruebas unitarias. Las herramientas de cobertura de código para pruebas unitarias se integran con las mismas, de modo tal que las métricas son generadas al final de la ejecución de las pruebas (sección 2.6.1).

PRÁCTICA 2.1.1: AGRUPAMIENTO DE PRUEBAS UNITARIAS

La categorización de pruebas se define como el proceso de agrupar o segmentar las pruebas según características en común. Para las pruebas unitarias, se utilizan dos grupos: módulos y componentes. En la Tabla 19 se presenta el objetivo de la práctica, los problemas que resuelve, los requisitos para implementarla y las tareas de implementación que la componen.

Tabla 19. Práctica: Agrupamiento de pruebas unitarias.

Agrupamiento de pruebas unitarias		
Objetivo	Agrupar las pruebas unitarias por módulos y funciones.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Baja mantenibilidad y poca escalabilidad del código de pruebas unitarias.	Poca cobertura de pruebas en Entrega Continua
	Imposibilidad de paralelización.	Tiempos de ejecución elevados.
	Imposibilidad de ejecución de pruebas para partes específicas del código.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continua**
	Código base agrupado estructuralmente por funcionalidad y módulo.	Existen pruebas unitarias automatizadas, dentro de una carpeta o directorio de pruebas.
Tareas	1. Agrupamiento de pruebas unitarias	

Tarea 2.1.1.1: Agrupamiento de pruebas unitarias

El agrupamiento de pruebas unitarias consiste en estructurarlas según el módulo y la funcionalidad que están verificando. El criterio debe ser el mismo que se utiliza para agrupar el código base de la aplicación. Por ejemplo, si el código es orientado a objetos, cumpliendo con el patrón de bajo acoplamiento y alta cohesión, el mismo estará estructurado en clases (criterio funcionalidad) y luego en paquetes (criterio módulo) de acuerdo con un determinado criterio. Para las pruebas, se debe seguir la misma estructura, de tal manera que cada clase/paquete del código base tendrá su correspondiente clase/paquete de prueba. Un pequeño ejemplo se muestra en la Fig. 44.

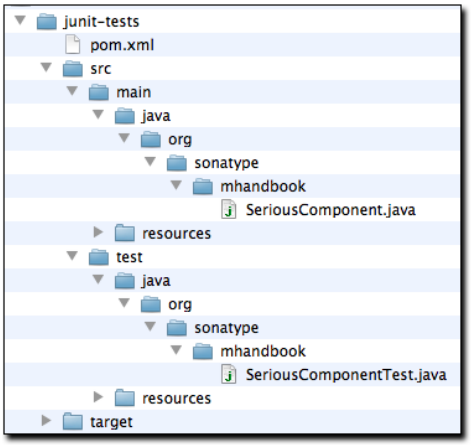


Fig. 44. Pruebas unitarias agrupadas según el código base.

PRÁCTICA 2.1.2: COBERTURA DE PRUEBAS UNITARIAS

La cobertura de pruebas unitarias es una medida utilizada para describir el grado en que el código fuente está cubierto por las pruebas unitarias. Un programa con una alta cobertura de pruebas, tiene una menor probabilidad de contener errores de software no detectados en comparación con un programa con baja cobertura de prueba [152]. Además, proporciona información crítica para mostrar a los equipos dónde enfocar las pruebas. Otra de las ventajas más importantes que presenta el uso de una herramienta para medir la cobertura de pruebas unitarias, es la posibilidad de detectar que parte del código ha sido modificada y, por ende, se puede reducir la cantidad de pruebas a ejecutar. Esta práctica se presenta en la

Tabla 20.

Tabla 20. Práctica: Cobertura de pruebas unitarias.

Cobertura de pruebas unitarias		
Objetivos	Detectar qué partes del código no se está probando y qué parte del código ha sido modificada.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Baja calidad del código fuente.	Poca cobertura de pruebas en Entrega Continua.
	Imposibilidad de ejecución de pruebas para partes específicas del código.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Existen pruebas unitarias automatizadas.
Tareas	1. Implementación de herramienta de cobertura de pruebas unitarias. 2. Definición del umbral de cobertura.	

Tarea 2.1.2.1: Implementación de herramienta de cobertura de pruebas unitarias

Dependiendo del lenguaje de programación utilizado tanto para la aplicación como para las pruebas unitarias, existen una gran cantidad de herramientas que pueden ser utilizadas para medir la cobertura del código. La mayoría de estas herramientas se integran al proyecto y su ejecución depende de la ejecución de las pruebas unitarias.

Una vez que las pruebas unitarias hayan finalizado su ejecución, la herramienta analiza el código y genera el reporte de cobertura. Independientemente de la herramienta y las características adicionales que presente, todas ellas muestran el nivel de cobertura del todo el proyecto, como así también de cada directorio y archivos. Algunas herramientas también muestran en detalle que partes del código están cubiertas y que partes no.

Tarea 2.1.2.2: Definición del umbral de cobertura

Si se define como objetivo un determinado nivel de cobertura, los desarrolladores intentarán alcanzarlo. El problema es que los números de alta cobertura son demasiado fáciles

de alcanzar con pruebas de baja calidad, como también puede existir lo que se conoce como pruebas sin aserciones.

Como la mayoría de los aspectos de la programación, las pruebas deben realizarse con mucha atención. Si se lo está haciendo, el porcentaje ideal sería de 80% o más [153]. Sería raro tener un 100%, ya que quizás signifique tener desarrolladores escribiendo pruebas para tener buenos números de cobertura, pero sin un criterio de eficiencia [153]. Ciertamente, los números de cobertura bajos (menores al 50%), son un signo de problemas

En resumen, es necesario establecer el umbral de cobertura o el nivel de cobertura deseado, para evitar desarrolladores que no escriban pruebas unitarias por cada cambio que introducen al repositorio de control de versiones.

4.2.2.2. Etapa de construcción

Dentro de esta etapa y para este nivel CTIL, no se tienen prácticas específicas de Prueba Continua, sino del desarrollo continuo en general. Como es un requisito para la próxima etapa de V&V, que las pruebas funcionales automatizadas se ejecuten inmediatamente después de la construcción del código, la aplicación debe estar en funcionamiento en un ambiente de prueba. Por lo tanto, en esta etapa se debe agregar un disparador a la tarea de despliegue del software y al finalizar, se debe invocar a la etapa de pruebas automatizadas. Además, es indispensable que el proceso de despliegue sea totalmente automatizado (sección 2.4).

PRÁCTICA 2.2.1: DESPLIEGUE AUTOMÁTICO

En un entorno continuo, parte de la misión es mejorar la eficiencia. La estandarización de los entornos de desarrollo y la automatización de los procesos de entrega respaldan esa misión. Además, las pruebas funcionales deben ejecutarse inmediatamente después de la construcción del código y esto depende del tiempo del despliegue. Los despliegues manuales pueden ser tediosos y llevar mucho tiempo [9]. Por lo general, hay muchos conjuntos específicos de tareas que deben realizarse meticulosamente en un orden específico. Es fácil, incluso para alguien que ha realizado el mismo despliegue 100 veces, olvidarse de la ejecución de un paso pequeño pero crítico. Cuando eso sucede, el problema puede ser obvio o puede llevar horas localizarlo. La solución es la implementación del despliegue automático. Esta práctica se presenta en la

Tabla 21.

Tabla 21. Practica: Despliegue automático**.**

Despliegue automático		
Objetivos	Automatizar el despliegue del software luego de la construcción de este.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Imposibilidad de ejecución de pruebas automáticamente luego de la construcción.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Pasos para el despliegue bien documentados.	No aplica.
Tareas	1. Automatización del despliegue.	

Tarea 2.2.1.1: Automatización e integración del despliegue

La automatización del despliegue es lo que permite poner en funcionamiento el software en entornos de prueba y producción con solo presionar un botón [10]. La automatización es esencial para reducir el riesgo de despliegues de producción. También es esencial para permitir que los equipos realicen pruebas exhaustivas tan pronto como sea posible después de los cambios (sección 2.3).

Una vez que el proceso de despliegue está automatizado en alguna tarea del servidor de Integración Continua, la misma debe ser disparada después de que las pruebas unitarias concluyan. Esto permitirá que el despliegue se realice de forma automática por cada cambio introducido en la rama principal del repositorio de control de versiones (sección 2.5.2).

4.2.2.3. Etapa de pruebas funcionales

Para esta etapa, el objetivo del nivel de mejora 2, consiste en organizar tanto el marco de trabajo como las pruebas funcionales automatizadas para aumentar la escalabilidad y permitir la

ejecución de pruebas utilizando prioridades. Esto también permitirá que, en el próximo nivel, las pruebas puedan ejecutarse solamente por grupos según sea necesario.

Finalmente, con el fin de brindar un entorno de pruebas para dispositivos móviles, se introduce la utilización de una granja de dispositivos móviles. Esta última práctica fue sugerida por los expertos que realizaron la validación en el apartado 6.1.4.

PRÁCTICA 2.3.1: IMPLEMENTACIÓN DE UN MODELO DE CAPAS PARA EL MARCO DE TRABAJO DE PRUEBAS FUNCIONALES

Escribir pruebas automatizadas mantenibles requiere, en primer lugar, una cuidadosa atención al proceso de análisis, ya que derivan de los requerimientos o criterios de aceptación. Una vez logrado eso, las pruebas se deben separar en tres capas [7]:

- La capa de más bajo nivel debe comprender cómo interactuar con la aplicación para realizar acciones y devolver resultados.
- En la capa intermedia, el código no interactúa con elementos de una GUI, llamadas de servicios o a otras partes de la capa lógica. En lugar de eso, invoca a las funciones brindadas por la capa de más bajo nivel.
- La capa de más alto nivel tiene lenguaje que es entendido por gente no técnica (interesados, dueños de producto, analistas, etc.).

La práctica que soporta la implementación de este modelo de capas se presenta en la Tabla 22.

Tabla 22. Practica: Implementación de un modelo de capas para el marco de trabajo de pruebas funcionales.

Implementación de un modelo de capas para el marco de trabajo de pruebas funcionales		
Objetivos	Crear y mantener pruebas automatizadas de forma eficaz.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Baja mantenibilidad: alto acoplamiento y baja cohesión de pruebas funcionales automatizadas.	Poca cobertura de pruebas en Entrega Continua.

Implementación de un modelo de capas para el marco de trabajo de pruebas funcionales		
	Personas sin conocimientos de código no pueden entender los reportes (reportes ambiguos).	Poca cobertura de pruebas en Entrega Continua.
	Pruebas frágiles que fallan debido a cambios intencionales en la capa lógica o en la interfaz gráfica de usuario.	Pruebas no deterministas.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Los requerimientos, criterios de aceptación y/o casos de pruebas se encuentran documentados.	Existe un marco de trabajo de automatización de pruebas funcionales, con herramientas implementadas para interactuar con la aplicación.
Tareas	1. Implementación de la capa del controlador de aplicación. 2. Refactorización de la capa de pruebas. 3. Implementación de la capa de criterios de aceptación.	

Tarea 2.3.1.1: Implementación de la capa del controlador de aplicación.

Las pruebas deben interactuar con la aplicación a través de una capa de más bajo nivel, la cual es denominada “capa de controlador de aplicación” [7]. Esta capa debe presentar una API que permita ejecutar acciones y retornar resultados.

Si las pruebas funcionales son de API, esta capa es la encargada de conocer los detalles de la API y realizar las llamadas a los servicios, generando las peticiones, procesando las respuestas y retornando los resultados finales. Un patrón utilizado para realizar esta implementación con pruebas de API, es el patrón de diseño para pruebas de servicios REST [154], [155]. Este patrón presenta la implementación de la capa de controlador de aplicación, llamada capa de servicios, que permite interactuar con servicios web, sin depender de un cliente en particular. También define el manejo de objetos de transferencia de datos en las peticiones y las respuestas. Por último, las pruebas solo interactúan con esta capa de servicios.

Por otro lado, si las pruebas funcionales son de GUI, esta capa debe contener un controlador de GUI, como Selenium WebDriver[46]. El patrón más utilizado para la implementación de esta capa con pruebas de GUI, es el modelo “Page Object” [156]. Este patrón se ha popularizado en la automatización de pruebas para mejorar el mantenimiento de las pruebas y reducir la duplicación de código, como se vio en la sección 3.2.2.3.

Tarea 2.3.1.2: Refactorización de la capa de pruebas.

En el CTIL 1. las pruebas funcionales comenzaron a automatizarse. Sin embargo, es probable que dentro de las mismas se haya agregado código para interactuar tanto con la GUI como con las API. Entonces, el objetivo de esta tarea es mover todo este código de interacción restante en las pruebas, a la capa de controlador de aplicación creada en la tarea anterior. Las pruebas que interactúan directamente con una API o GUI son frágiles y producen pruebas no deterministas (sección 3.5).

Tarea 2.3.1.3: Implementación de la capa de criterios de aceptación.

La capa de más alto nivel para las pruebas funcionales automatizadas, es la capa de criterios de aceptación (sección 2.6) [7]. Los analistas definen los criterios de aceptación para las historias de usuario, criterios que deben cumplirse para que la historia de usuario sea reconocida como hecha. Chris Matts y Dan North idearon un lenguaje específico de dominio para escribir criterios de aceptación, que toma la siguiente forma [157]:

Dado un contexto inicial,
Cuando un evento se produce,
Entonces existen salidas.

Herramientas como Cucumber, JBehave, Concordion, Twist o FitNesse permiten poner criterios de aceptación directamente en las pruebas y vincularlos a la implementación subyacente. También se puede adoptar el enfoque de codificar los criterios de aceptación en los nombres de las pruebas utilizando el marco de trabajo de pruebas. Luego se pueden ejecutar las

[46] Selenium WebDriver es una API que mediante ciertos comandos permite interactuar con un navegador web de modo que se pueda simular el uso de una aplicación web por parte del usuario final. https://selenium.dev/projects/

pruebas directamente desde el marco de trabajo de pruebas, en lugar de una capa de criterios de aceptación. Esto ha sido una mejora añadida en el tercer ciclo de IA (sección 6.2.4.4). Finalmente, utilizando herramientas como AgileDox[47], es posible generar documentación a partir de las funciones o métodos de pruebas.

PRÁCTICA 2.3.2: SEGMENTACIÓN DE PRUEBAS FUNCIONALES

La categorización de pruebas se define como el proceso de agrupar o segmentar las pruebas según características en común. Para las pruebas funcionales, se utilizan dos grupos: funcionalidades y prioridad. En primer lugar, si las pruebas están agrupadas por funcionalidad, se tiene la posibilidad de ejecutar solo determinadas pruebas para cambios pequeños, en lugar del lote completo. Por otro lado, si las pruebas se encuentran categorizadas según su prioridad (pruebas de humo, regresión, etc.), se pueden ejecutar las más críticas al principio, permitiendo la detección de fallos críticos lo antes posible. Una ventaja adicional es que la paralelización es más fácil de implementar teniendo pruebas segmentadas.

El objetivo de la segmentación de pruebas funcionales, junto con los problemas que resuelve y las tareas para implementarla, se detallan a continuación Tabla 23.

Tabla 23. Practica: Segmentación de pruebas funcionales.

Segmentación de pruebas funcionales		
Objetivo	Agrupar las pruebas funcionales según funcionalidad y prioridad en su ejecución.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Baja Mantenibilidad: Poca escalabilidad del código de pruebas funcionales.	Poca cobertura de pruebas en Entrega Continua.
	Imposibilidad de paralelización.	Tiempos de ejecución elevados.
	Imposibilidad de ejecución de pruebas para partes específicas del código.	
	Imposibilidad de priorización.	
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Existen pruebas funcionales automatizadas.

[47] http://agiledox.sourceforge.net/

Segmentación de pruebas funcionales		
		El marco de trabajo de pruebas utilizado soporta la funcionalidad de agrupamiento.
Tareas	1. Relevamiento de las funcionalidades generales y más críticas del sistema. 2. Agrupamiento de pruebas funcionales.	

Tarea 2.3.2.1: Relevamiento de las funcionalidades generales y más críticas del sistema.

Esta tarea es un paso dentro de la definición y establecimiento de la política de pruebas en una organización o proyecto de desarrollo. De hecho, el TMMI determina que para establecer la política de pruebas se deben definir los objetivos de pruebas, tomando como base las necesidades de la organización, sus objetivos, las áreas más importantes del sistema, etc. [105]. Para ello, se deben realizar todas las reuniones que sean necesarias con los interesados en el sistema y el dueño del producto, para relevar con exactitud estos objetivos y generar una traza con las necesidades de la organización (sección 2.2). El próximo paso es la definición de la estrategia de pruebas [105].

Durante las reuniones que se realizan en el marco del establecimiento y definición de la política y la estrategia de pruebas, se debe realizar un relevamiento de las principales funcionalidades del sistema, documentarlos y publicarlos para que esté disponible para todo el equipo de desarrollo, interesados en el sistema, jefes de proyecto, etc. También se deben determinar durante estas reuniones cuáles son las funcionalidades más importantes para el negocio utilizando una escala de categorización de severidad, por ejemplo, de tres niveles: alta, media, baja. El resultado es una tabla de funcionalidades y niveles de criticidad, como el ejemplo que se muestra en la Tabla 24. La escala de severidad dependerá de la implementación y buscará evitar sesgos, tal y como se describe en [7].

Tabla 24. Tabla de funcionalidades y niveles criticidad de la página de una aerolínea.

Funcionalidad	Criticidad
Consulta de vuelos	Alta
Compra del vuelo	Alta
Franquicia de equipaje	Baja

Servicios adicionales a la reserva	Media
Consulta de estado de vuelo	Media
Promociones	Media
Sucursales y Aeropuertos	Baja

En algunos proyectos, se da el caso que para una misma funcionalidad existen escenarios donde algunos son de criticidad alta y otros no. Por ejemplo, utilizando el caso anterior de la aerolínea, el proceso de compra del vuelo puede ser una funcionalidad de criticidad alta, pero quizás con determinados parámetros como un número de pasajeros muy alto, algunas rutas de vuelo poco frecuentadas, algún rango de fechas muy lejano, etc., los escenarios se convierten en casos de prioridad media o baja. Es de suma importancia agregar todas estas aclaraciones en la tabla de funcionalidades y niveles de criticidad.

Finalmente, a medida que el proyecto avanza y se agregan funcionalidades, las mismas deben ser agregadas en la tabla, con su correspondiente análisis de escenarios y nivel de criticidad, definidos por los interesados en el sistema o el dueño del producto.

Tarea 2.3.2.2: Segmentación de pruebas funcionales.

Esta tarea consiste en recorrer todas las pruebas funcionales automatizadas y marcarlas como pertenecientes a una funcionalidad y a un nivel de criticidad. Para ello, la mayoría de los marcos de trabajo de pruebas tienen una característica de agrupamiento, utilizando anotaciones o etiquetas. La funcionalidad y el nivel de criticidad a utilizar en cada prueba automatizada se toma del relevamiento realizado en el paso anterior.

Posteriormente, se deben crear archivos de configuración de lotes de pruebas. Un archivo de lotes de pruebas permite ejecutar casos de pruebas automatizados pertenecientes a uno o varios grupos específicos. También permite excluir casos de pruebas de otros grupos. El objetivo es crear lote por cada funcionalidad y por cada nivel de criticidad.

Finalmente, el último paso es la priorización, la cual consiste en configurar el servidor de Integración Continua, para que primero ejecute las pruebas de criticidad alta (pruebas de humo), luego las pruebas de criticidad media y por último las pruebas de criticidad baja. Esto se realiza mediante el uso de las suites creadas por cada nivel de criticidad.

PRÁCTICA 2.3.3: GRANJA PARA PRUEBAS DE DISPOSITIVOS MÓVILES

Una granja de dispositivos consiste en un conjunto de dispositivos móviles reales conectados a un servidor, los cuáles pueden ser accedidos de forma remota mediante el acceso a la red donde se encuentra. La ventaja frente a los servicios de dispositivos móviles en la nube es que permite enfocarse solamente en los dispositivos soportados por la aplicación, los cuales a veces no están disponibles en la nube. Además, se evitan los costos que se tienen con la ejecución de regresiones de forma repetida en estos servicios. Por otro lado, también es necesario el uso de dispositivos reales, ya que a veces es importante verificar si el software funciona en ellos (y no solo en simuladores), incluido el problema de la carga potencial intensiva de CPU y GPU, para revelar problemas de bajo consumo o de bajo rendimiento de la batería.

La práctica que guía la implementación de una granja para pruebas de dispositivos móviles se muestra en la Tabla 25.

Tabla 25. Practica: Granja para pruebas de dispositivos móviles.

Granja para pruebas de dispositivos móviles		
Objetivo	Brindar un entorno de pruebas para sitios web responsivos y aplicaciones en un número significativo de dispositivos móviles.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Falta de confiabilidad de las pruebas en dispositivos móviles emulados o simulados.	Poca cobertura de pruebas en Entrega Continua.
	Costos elevados.	
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	La aplicación que se prueba es soportada en dispositivos móviles, o es responsiva.	Existen casos de pruebas funcionales para dispositivos móviles.
Tareas	1. Implementación de la granja de dispositivos móviles.	

Tarea 2.3.3.1: Implementación de la granja de dispositivos móviles.

El enfoque estándar para organizar pruebas de software en este tipo de plataformas es conectar dispositivos móviles a un servidor que ejecute algún software especial [158]. Para las pruebas automatizadas una máquina remota inicia la ejecución de las pruebas (cliente), realizando la petición al servidor para conectarse a los dispositivos móviles. Cuando finalizan, se pueden obtener reportes de esta ejecución. Los elementos principales para la granja son:

- Un software de gestión y ejecución de pruebas para dispositivos móviles: hay muchas soluciones, pero la más conocida es Appium[48].
- Servidor de pruebas: una PC en donde se ejecuten las pruebas automatizadas. La misma debe tener acceso a los servidores de dispositivos móviles.
- Servidor de dispositivos móviles Android: una PC ejecutando el software de pruebas, conectado a los dispositivos móviles Android.
- Servidor de dispositivos móviles IOs (en caso de ser necesario): una PC con OSx ejecutando el software de pruebas, conectado a los dispositivos móviles IOs.
- Dispositivos móviles necesarios.

Un ejemplo de esta arquitectura utilizando Selenium como servidor de pruebas y Appium para los servidores de dispositivos móviles, se presenta en la Fig. 45.

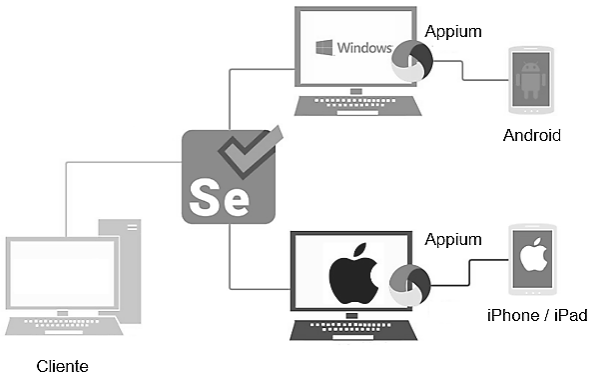


Fig. 45. Arquitectura simplificada de una granja de dispositivos.

El cliente puede ser un usuario (probador manual), un servidor de Integración Continua, o una computadora ejecutando pruebas funcionales automatizadas para los dispositivos móviles.

4.2.2.4. Etapa de pruebas no funcionales

Al igual que en las otras etapas de este nivel, el objetivo es organizar las pruebas relacionadas. En este caso, las pruebas son las de rendimiento, las cuales se utilizan para detectar anomalías en los tiempos de respuesta del sistema. Sin embargo y a partir de sugerencias realizadas por los expertos que participaron de la evaluación del modelo (sección 6.1.4), es necesario organizar este proceso utilizando un enfoque de tres etapas:

[48] http://appium.io/

1. Implementación de pruebas sobre características más específicas, denominadas pruebas de capacidad.
2. Establecimiento de niveles de aceptación para este tipo de pruebas. Estos niveles se conocen como umbrales de rendimiento o capacidad.
3. Utilización de un entorno similar a producción, para aumentar la confiabilidad de los resultados.

PRÁCTICA 2.4.1: GESTIÓN DE PRUEBAS DE CAPACIDAD

El siguiente paso luego de comenzar a medir el rendimiento de una aplicación, es organizar este proceso mediante la definición de pruebas de capacidad, estableciendo los umbrales deseados e implementando un entorno de pruebas de rendimiento [7]. Esta gestión de pruebas de capacidad se presenta como práctica en la Tabla 26. Las pruebas de capacidad abarcan gran cantidad de características del rendimiento del sistema, como escalabilidad, longevidad, cantidad de transacciones por segundo, carga, etc. Posteriormente, el establecimiento de los umbrales es necesario para determinar cuándo el resultado de la ejecución de una prueba de capacidad ha fallado. Finalmente, utilizar un entorno de pruebas de capacidad es importante para evitar fallos no deseados y añadir confiabilidad a las pruebas.

Tabla 26. Practica: Gestión de pruebas de capacidad.

Gestión de pruebas de capacidad		
Objetivo	Gestionar adecuadamente el proceso de pruebas de rendimiento, mediante la implementación de diferentes tipos de pruebas de capacidad en un entorno especializado.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Falta de confiabilidad de las pruebas en entornos diferentes a producción.	Poca cobertura de pruebas en Entrega Continua.
	Falsos positivos por errores de red o de ambiente.	Pruebas no deterministas.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Tener como objetivo evitar que la aplicación colapse, debido a problemas de carga elevada.	Se mide el rendimiento de la aplicación.

Gestión de pruebas de capacidad	
Tareas	1. Definición de pruebas de capacidad. 2. Establecimiento de umbrales. 3. Implementación del entorno de pruebas.

Tarea 2.4.1.1: Definición de pruebas de capacidad.

Esta tarea consiste en determinar qué tipos de pruebas de capacidad son adecuados para el proyecto (sección 2.6.4.2). Para determinarlo, se deben responder a las siguientes preguntas: ¿cuántas transacciones por segundo puede almacenar la base de datos?, ¿cuántos mensajes por segundo puede transmitir la cola de mensajes? Incluso, se pueden relacionar a preguntas de negocio, como, por ejemplo: ¿Cuántas ventas por segundo se pueden hacer en un determinado momento del día?, ¿Puede el usuario utilizar el sistema eficientemente durante picos de carga? Una vez respondidas esas preguntas, se escogen las pruebas de capacidad a utilizar.

Tarea 2.4.1.2: Establecimiento de umbrales.

La importancia de las pruebas de capacidad son la medición de los resultados y el éxito o el fallo se determina mediante un análisis humano de estas mediciones. Esta puede ser una tarea subjetiva, sin embargo, la capacidad de generar mediciones y proporcionar información sobre lo que sucedió, es más útil que tener solo un informe binario de falla o éxito [15]. Por esta razón, es importante generar gráficos complementarios en los reportes, para la toma de decisiones junto con el análisis de los valores absolutos. Un ejemplo de gráfico generado con la herramienta Gatling[49] se muestra en la Fig. 46. Estos gráficos deben ser fácilmente accesibles desde los tableros visuales del Conducto de Despliegue.

[49] https://gatling.io/

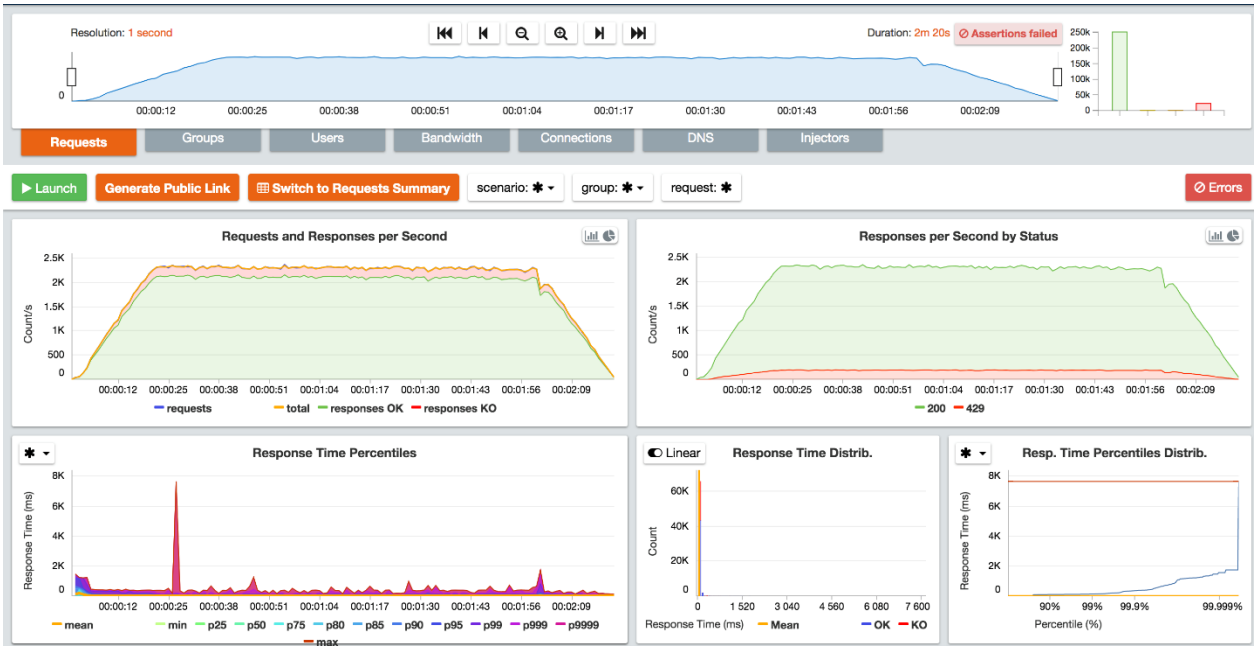


Fig. 46. Reporte gráfico de pruebas de capacidad.

Para cada prueba que se ejecuta, también se deberá definir la interpretación de un resultado como “éxito” o “fallo” si se establece un nivel demasiado alto, es probable obtener resultados de ejecución regulares e intermitentes. Las pruebas pueden fallar cuando la red está en uso para otras tareas, o cuando el entorno donde se ejecutan las pruebas de capacidad está trabajando simultáneamente en otra tarea. Por otro lado, con un nivel muy bajo, no se detectará la introducción de un cambio que pueda reducir el rendimiento, por ejemplo, verificando 100 transacciones por segundo cuando la aplicación soporta realmente 200.

Entonces, para establecer los umbrales se toman dos estrategias [7]:

- Buscar un objetivo estable y reproducible. En la medida de lo posible, se debe tener un ambiente de prueba de capacidad aislado de otras influencias. Esto minimiza el impacto de otras tareas no relacionadas con las pruebas y, por lo tanto, hace que los resultados sean más consistentes. La prueba de capacidad es una de las pocas situaciones en las que la virtualización no es apropiada (a menos que el entorno de producción sea virtual) debido a la sobrecarga que introduce para el rendimiento.
- Ajustar el umbral de aceptación para cada prueba incrementándolo una vez que la prueba pase a un nivel mínimo aceptable. Esto brinda protección contra el escenario de falso positivo. Si la prueba comienza a fallar después de la introducción de un cambio y el umbral se establece muy por encima del requerimiento, se puede simplemente bajar el umbral si la degradación de la capacidad es por una razón bien comprendida y aceptable,

pero la prueba mantendrá su valor como protección contra cambios involuntarios que dañan la capacidad.

Finalmente, para que las pruebas sean genuinas, en lugar de mediciones de rendimiento, cada una debe incorporar un escenario específico y debe evaluarse contra un umbral más alto del que se considera para que la prueba sea exitosa [78].

Tarea 2.4.1.3: Implementación del entorno de pruebas.

Las mediciones absolutas de la capacidad de un sistema deberían realizarse idealmente en un entorno que, lo más fielmente posible, reproduzca el entorno de producción en el que el sistema finalmente se ejecutará [7]. Esta tarea fue añadida a la tercera versión del modelo, luego de implementar el segundo ciclo de validación (sección 6.2.3.4)

Si la capacidad o el rendimiento son problemas graves para la aplicación, entonces es recomendable realizar una inversión para crear una réplica del entorno de producción para las partes centrales del sistema. Esto implica usar las mismas especificaciones de hardware y software y la misma configuración para cada entorno, incluyendo la configuración de redes, la lógica de intercambio entre aplicaciones (*middleware*) y sistema operativo.

Una estrategia para limitar los costos del entorno de prueba y proporcionar algunas medidas de rendimiento precisas, es replicar una porción de los servidores de producción de forma porcentual. Por ejemplo, si en producción se tienen 4 servidores de aplicación, 2 servidores de base de datos y 2 servidores web, entonces el ambiente de capacidad podría estar formado por 2 servidores de aplicación, 1 servidor de base de datos y 1 servidor web. Esta infraestructura es presentada en la Fig. 47.

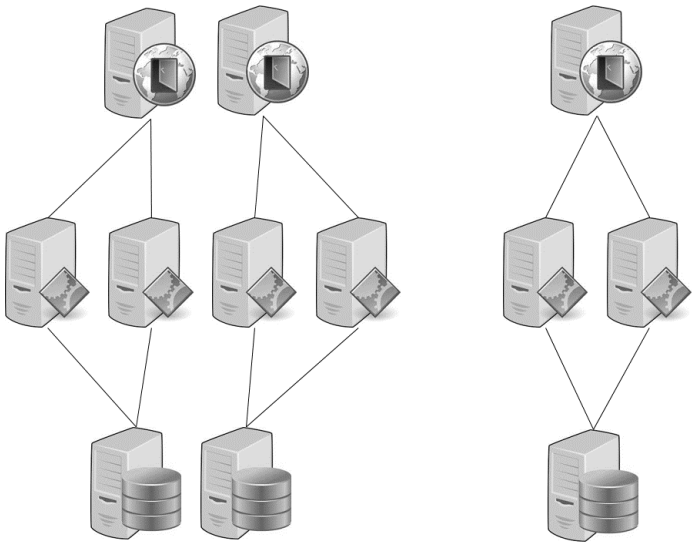


Fig. 47. Replica de un ambiente de capacidad, con respecto al ambiente de producción.

De este modo, se puede extrapolar la capacidad considerando la diferencia que se consideró para implementar el ambiente de capacidad, con respecto al ambiente de producción.

4.2.2.5. Etapa de pruebas de aceptación de usuario

La mayoría de las pruebas de calidad de software utilizan un enfoque estructurado [78]. Los casos de pruebas se definen en base a historias de usuario ya definidas y los datos de prueba se estructuran en función de los mismos. Esto hace que se pierdan de vista escenarios especiales, límites, o poco frecuentes, los que se descubren a través de pruebas exploratorias. Estas son de naturaleza aleatoria y pueden revelar errores que no se descubrirían con pruebas estructuras de aceptación [79]. En este nivel CTIL, esta etapa comprende la implementación y gestión de pruebas exploratorias.

Con las pruebas exploratorias, los especialistas en pruebas pueden ejecutar escenarios alternativos y negativos, detectando defectos y creando documentación sobre la marcha [7].

PRÁCTICA 2.5.1: IMPLEMENTACIÓN Y GESTIÓN DE PRUEBAS EXPLORATORIAS

La ejecución de pruebas de aceptación permite verificar que la funcionalidad que se está desarrollando cumple con los requerimientos principales. Sin embargo, esto no garantiza que la misma no esté libre de defectos, por lo que debe complementarse con otros enfoques [159]. Esto lleva a que las mismas deban complementarse con la ejecución de escenarios alternativos y

negativos [11], los cuales también deben ser documentados. Estas pruebas se conocen como pruebas exploratorias y la práctica que soporta su implementación y gestión se presenta en la Tabla 27.

Tabla 27. Practica: Implementación y gestión de pruebas exploratorias.

Implementación y gestión de pruebas exploratorias		
Objetivo	Implementar una estrategia de pruebas manuales exploratorias.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Falta de cobertura en pruebas manuales.	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	No aplica.
Tareas	1. Implementación de la estrategia de pruebas manuales exploratorias.	

Tarea 2.5.1.1: Implementación de la estrategia de pruebas manuales

Una de las actividades de esta etapa para el nivel 1 de este modelo, eran las pruebas manuales de aceptación, donde se verifican que los requerimientos se cumplan. En este nivel CTIL, esta etapa debe abarcar los casos de pruebas críticos que no han sido automatizados en la etapa de pruebas funcionales y no funcionales.

Por otro lado, también debe considerase la ejecución de escenarios alternativos y negativos [78]. Por ejemplo, si se está verificando un campo que acepta solo números del 1 al 5, el mismo deberá ser probado con números fuera de ese rango, caracteres alfabéticos, caracteres especiales, espacios en blanco, etc. Estas pruebas se denominan exploratorias, ya que no se encuentran atadas a casos de pruebas de requerimientos específicos y lo que se hace es "explorar" la aplicación en busca de defectos.

Las pruebas alternativas y negativas no afectan solamente a la GUI, también pueden ejecutarse sobre el la lógica de negocio [20]. Por ejemplo, un servicio REST que devuelve un conjunto de países cuando se realiza una petición que contiene como parámetro un continente, puede devolver un error cuando el parámetro es distinto a un continente válido (continente incorrecto, tipo de dato diferente, etc.).

Este conjunto de escenarios adicionales a ejecutar debe ser parte de la estrategia de pruebas definida en etapas anteriores y también deben estar documentados como casos de pruebas. Esto permitirá, por un lado, disminuir la posibilidad de introducir errores en producción,

mediante la detección de defectos no encontrados con las pruebas de aceptación. Por otro lado, estos escenarios documentados pueden ser automatizados en el futuro. Finalmente, la aparición de un defecto durante las pruebas exploratorias, debe ser la entrada para la creación de nuevos casos de pruebas y/o requerimientos (sección 3.2.2.3).

4.2.2.6. Análisis del nivel de mejora 2 y los problemas en Pruebas Continuas

El nivel CTIL 2 se implementan buenas prácticas relacionadas principalmente con la gestión de las pruebas. Las pruebas unitarias y las funcionales son agrupadas. Se mide la cobertura de las pruebas unitarias y el marco de trabajo de pruebas funcionales se divide en capas para facilitar su mantenibilidad. También se propone la utilización de una granja para las pruebas en dispositivos móviles. Finalmente, se reemplaza la medición de tiempos de respuesta por pruebas de capacidad y se añaden pruebas exploratorias manuales para detectar casos de borde. El conducto de Pruebas Continuas formado por estas buenas prácticas sumadas a las recomendaciones del nivel 1, se presenta en la Fig. 48.

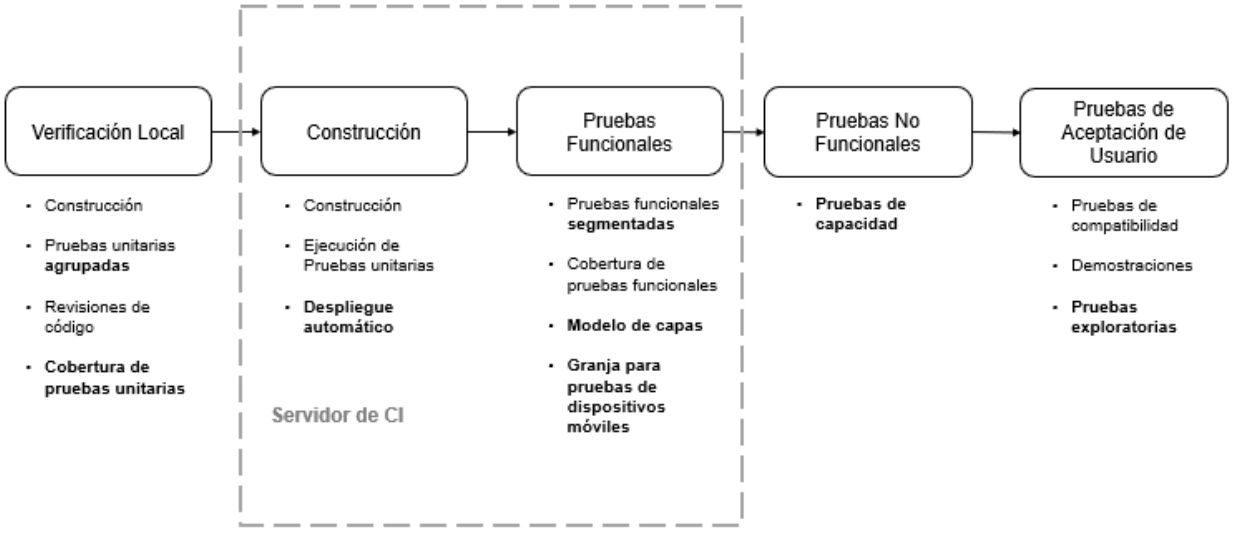


Fig. 48. Conducto de Pruebas Continuas en el nivel de mejora 2.

Nuevamente, la cobertura de pruebas en Entregas Continua aumenta a un 64%, donde 28% se logra con la implementación de este nivel (7 problemas de 25) más el 36% obtenido con el nivel 1. Por otro lado, los problemas de pruebas que consumen mucho tiempo de ejecución también se atacan en un 22% (4 problemas de 18), que sumado al 11,11% logrado en el nivel 1, genera un 33,33. Finalmente, las pruebas no deterministas comienzan a solucionarse en un 22,22% (2 problemas de 9). Esto se muestra en la Fig. 49.

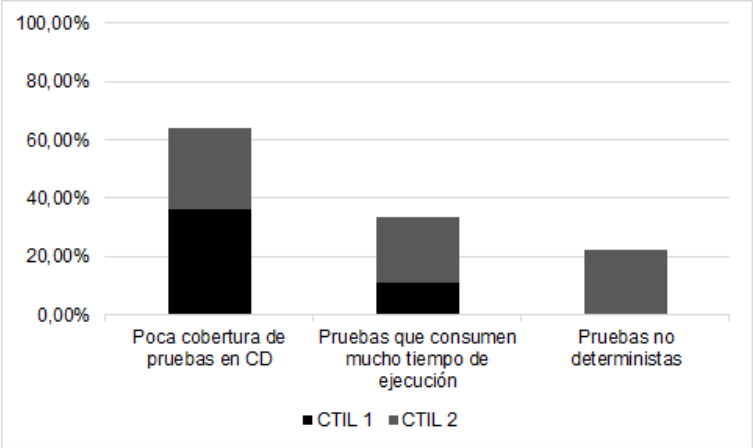


Fig. 49. Grado de resolución de los problemas de Pruebas Continua al implementar el nivel 2 del modelo.

4.2.3. Nivel de Mejora 3

Como se menciona al inicio de este modelo (sección 4.2), en el nivel de mejora 3 se definen estándares y patrones para el proceso de pruebas, y, en consecuencia, la etapa de ejecución de pruebas genera resultados confiables. La confiabilidad es otro requisito fundamental de las Pruebas Continuas, ya que el lanzamiento del software a producción con solo presionar un botón, requiere de pruebas confiables que aseguren la no introducción de errores [7], [10]. La confiabilidad no solo significa pruebas deterministas, sino también mayor cobertura en su ejecución.

En esta etapa, el objetivo es aumentar la confiabilidad de las pruebas, mediante la incorporación de verificaciones más exhaustivas en los diferentes niveles de pruebas [78], [79], [82], [83], el aumento en la cobertura de las mismas y la eliminación de los falsos positivos y de los resultados ambiguos.

4.2.3.1. Etapa de Verificación local

En esta etapa de V&V, se busca aumentar la cobertura de las pruebas unitarias, mediante la incorporación de verificaciones de código que interactúa con sistemas externos. Para ello, se utiliza la práctica de simulación de objetos y resultados [160], [161].

Además, para aumentar la confiabilidad de las verificaciones en esta etapa antes de introducir los cambios al repositorio, se propone la ejecución de pruebas funcionales

automatizadas contra la versión funcional del código en el entorno local del desarrollador. Esta última práctica fue incorporada luego de la etapa de validación (apartado 6.1.4).

PRÁCTICA 3.1.1: USO DE PRUEBAS SIMULADAS

Una pieza fundamental de las pruebas automatizadas es el reemplazo de partes del sistema en tiempo de ejecución con una versión simulada [162]. Esto se debe a que la automatización de verificaciones con componentes externos es compleja y si no se realiza entonces la cobertura del código no brindará confiabilidad. De este modo, se simulan estos componentes[50] [163].

La Tabla 28 presenta la práctica que propone el uso de pruebas simuladas.

Tabla 28. Practica: Uso de pruebas simuladas.

Uso de pruebas simuladas		
Objetivo	Permitir la creación de pruebas unitarias que interactúan con otros componentes externos como bases de datos, sistemas de archivos, o interfaces, cuya implementación es muy compleja. Esto se logra mediante la simulación de esos componentes con los que el código interactúa.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Complejidad del código.	Poca cobertura de pruebas en Entrega Continua.
	Falta de cobertura de código que interactúa con componentes externos.	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	El código a verificar interactúa con componentes externos (bases de datos, sistemas de archivos, etc.)	Existen pruebas unitarias automatizadas.
		Existe una herramienta que mide la cobertura de las pruebas unitarias.
Tareas	1. Instalación de una herramienta de simulación de pruebas. 2. Identificación de código sin cobertura que interactúa con componentes externos.	

[50] *Mocks y Stubs*

Uso de pruebas simuladas	
	3. Implementación de pruebas simuladas.

Tarea 3.1.1.1: Instalación de una herramienta de simulación de pruebas

El primer paso es agregar al proyecto una herramienta que permita la construcción de *mocks* y *stubs*. Existen una gran cantidad de herramientas disponibles según el lenguaje de programación utilizado. También varían según el tipo de componente que se quiera simular (API REST, base de datos, etc.).

Entre las herramientas más conocidas se encuentran Mockito (para Java), Rhino (para C#), Moq (para C#), jsmockito (para javascript), etc.

Una vez integrada la herramienta al proyecto, el próximo paso es la detección del código que carece de cobertura e interactúa con componentes externos [161].

Tarea 3.1.1.2: Identificación de código sin cobertura que interactúa con componentes externos

Utilizando la herramienta de cobertura de pruebas unitarias, es posible determinar qué partes del código no están siendo verificadas. En este grupo se encuentran algunas funciones que interactúan con componentes externos como las bases de datos.

Para este grupo de funciones, se deben utilizar las pruebas simuladas[51]. Entonces, se debe generar un listado de las funciones o métodos más críticos que interactúan con componentes externos para luego desarrollar las pruebas unitarias correspondientes.

Tarea 3.1.1.2: Implementación de pruebas simuladas

[51] Test Doubles

```
import java.sql.Connection;
import java.sql.PreparedStatement;
import java.sql.ResultSet;
import java.sql.SQLException;
import javax.sql.DataSource;
import org.anvard.introtojava.Person;
import org.springframework.util.Assert;

public class PersonDao {

      private DataSource ds;

      public PersonDao(DataSource ds) {
    this.ds = ds;
  }

  public void create(Person person) {
    try {
      Connection c = ds.getConnection();
      PreparedStatement stmt = c
                  .prepareStatement("INSERT INTO person (id, first_name,
                        last_name) values (?, ?, ?)");
      stmt.setInt(1, person.getId());
      stmt.setString(2, person.getFirstName());
      stmt.setString(3, person.getLastName());
      stmt.executeUpdate();
      c.close();
    } catch (SQLException e) { throw new DataAccessException(e); }
  }

  public Person retrieve(int id) {
    try {
      Connection c = ds.getConnection();
      PreparedStatement stmt = c
                  .prepareStatement("SELECT id, first_name,
                        last_name FROM person WHERE id = ?");
      stmt.setInt(1, id);
      ResultSet rs = stmt.executeQuery();
      if (!rs.first()) return null;
      Person p = new Person(rs.getInt(1), rs.getString(2),
                  rs.getString(3));
      c.close();
      return p;
    } catch (SQLException e) { throw new DataAccessException(e); }
  }
}
```

Algoritmo 4. Clase a verificar con pruebas simuladas.

Una vez que se tenga el listado mencionado en la tarea anterior, se comienza gradualmente a implementar las pruebas simuladas. Por un lado, primero se programan los *mocks* con los resultados esperados que forman la especificación de las llamadas de otro método o función que se espera que reciban. Pueden lanzar una excepción si reciben una llamada que

no esperan. Sin embargo, es muy fácil hacer un mal uso de los *mocks* escribiendo pruebas que no tienen sentido o son frágiles [7], usándolas simplemente para afirmar los detalles específicos del funcionamiento de algún código en lugar de sus interacciones con los componentes externos. Tal uso es frágil porque si la implementación cambia, la prueba se rompe.

Por otro lado, también se implementan los *stubs* que proporcionan respuestas enlatadas a las llamadas realizadas durante la prueba. En el Algoritmo 4 se presenta un ejemplo de prueba simulada usando Mockito[52]. El código que se va a verificar con la prueba simulada es una clase que accede a la base de datos, utilizando JDBC, para ejecutar comandos simples SQL.

Lo que hace que el Algoritmo 4 sea un desafío para las pruebas es que hay múltiples interfaces involucradas. La interfaz *DataSource* es utilizada para obtener una *Connection*. Luego, la interfaz *Connection* se usa para obtener una clase *PreparedStatement* y por último *PreparedStatement* se usa para generar el *ResultSet*. Sin embargo, usando las pruebas simuladas, se pueden simular cada uno de estos elementos. La prueba final se presenta en el Algoritmo 5.

Es importante que las pruebas unitarias que ya fueron desarrolladas para componentes externos sin la utilización de pruebas simuladas sean refactorizadas para utilizarlos. Esto permitirá que las pruebas sean más robustas, menos complejas y más fáciles de comprender.

PRÁCTICA 3.1.2: EJECUCIÓN DE PRUEBAS FUNCIONALES EN MEMORIA

La rama principal no puede contener código que genere errores [9]. Una de las maneras de evitarlo, es quitando los últimos cambios introducidos, una vez que las pruebas detectan los errores en el servidor de Integración Continua. Sin embargo, mientras las pruebas se ejecutan en el servidor de Integración Continua, otros desarrolladores pudieron descargarse la última versión del código, antes de que estas terminen y detecten los errores. De esta manera, esta práctica propone la ejecución de las pruebas funcionales más importantes antes de realizar la integración del código a la rama principal, para detectar los errores antes que el servidor de Integración Continua. Esta práctica de ejecución de pruebas funcionales en memoria se presenta en la Tabla 29.

[52] https://site.mockito.org/

```
import static org.junit.Assert.*;
import static org.mockito.Matchers.*;
import static org.mockito.Mockito.*;
import java.sql.*;
import javax.sql.DataSource;
import org.anvard.introtojava.Person;
import org.junit.*;
import org.mockito.Mock;
import org.mockito.runners.MockitoJUnitRunner;

@RunWith(MockitoJUnitRunner.class)
public class PersonDaoTest {

  @Mock
  private DataSource ds;
  @Mock
  private Connection c;
  @Mock
  private PreparedStatement stmt;
  @Mock
  private ResultSet rs;

  private Person p;

  @Before
  public void setUp() throws Exception {
    when(c.prepareStatement(any(String.class))).thenReturn(stmt);
    when(ds.getConnection()).thenReturn(c);
    p = new Person(1, "Johannes", "Smythe");
    when(rs.first()).thenReturn(true);
    when(rs.getInt(1)).thenReturn(1);
    when(rs.getString(2)).thenReturn(p.getFirstName());
    when(rs.getString(3)).thenReturn(p.getLastName());
    when(stmt.executeQuery()).thenReturn(rs);
  }

  @Test(expected=IllegalArgumentException.class)
  public void nullCreateThrowsException() {
    new PersonDao(ds).create(null);
  }

  @Test
  public void createPerson() {
    new PersonDao(ds).create(p);
  }

  @Test
  public void createAndRetrievePerson() throws Exception {
    PersonDao dao = new PersonDao(ds);
    dao.create(p);
    Person r = dao.retrieve(1);
    assertEquals(p, r);
  }
}
```

Algoritmo 5. Prueba simulada usando Mockito.

Tabla 29. Practica: Ejecución de pruebas funcionales en memoria.

Ejecución de pruebas funcionales en memoria		
Objetivo	Aumentar la confiabilidad de las verificaciones en esta etapa mediante ejecución de pruebas funcionales automatizadas en memoria antes de integrar el código a la rama principal.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Posible introducción en la rama principal de código que genere errores en la aplicación.	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	El código a verificar interactúa con componentes externos (bases de datos, sistemas de archivos, etc.)	Existen pruebas unitarias automatizadas.
		Existe una herramienta que mide la cobertura de las pruebas unitarias.
Tareas	1. Ejecución de pruebas funcionales antes de integrar el código. 2. Configuración de pruebas funcionales en memoria.	

Tarea 3.1.2.1: Ejecución de pruebas funcionales antes de integrar el código

Como se ha descrito en la sección 2.6.4.1 existen dos maneras de ejecutar las pruebas funcionales antes de que el código sea integrado en la rama principal: pruebas funcionales en el ambiente local del desarrollador y pruebas funcionales en el servidor de Integración Continua utilizando una rama funcional.

Antes de comenzar a ejecutar estas pruebas, es necesario identificar qué áreas de la aplicación fueron modificadas. Esto es importante ya que, si se ejecutan todas las pruebas funcionales automatizadas, se introducirá mucho tiempo en la etapa de verificación local y por ende a los ciclos de desarrollo. Otro problema es que pueden generar momentos en el que desarrollador se encuentre ocioso. Por esta razón, solo se deben ejecutar, en primer lugar, las pruebas más prioritarias (pruebas de humo) y posteriormente, las pruebas relacionadas al área de la aplicación que fue modificada por el desarrollador.

Tarea 3.1.2.2: Configuración de pruebas funcionales en memoria

Una vez que la ejecución de pruebas funcionales fue automatizada como una actividad antes de integrar los cambios a la rama principal, surgirá la necesidad de ejecutar la mayor

cantidad de pruebas lo más rápido posible para dar retroalimentación. Cómo el objetivo de la ejecución de las pruebas funcionales en este nivel es detectar errores críticos antes de integrar el código, no es necesario utilizar navegadores reales, sino un “navegador en memoria”.

Un navegador en memoria es un navegador que no tiene una GUI y como su nombre lo indica, se ejecuta solo en la memoria del SO en segundo plano. La mayoría de las herramientas de automatización de pruebas tienen soporte para navegadores en memoria. Por ejemplo, en Selenium, la creación del mismo se realiza utilizando la clase “HTMLUnitDriver”[53], como se muestra en el Algoritmo 6.

```
public class BrowserFactory {

  public static WebDriver createDriver(String type) {
    switch (type) {
      case "chrome":
        return new ChromeDriver();
      case "firefox":
        return new FirefoxDriver();
      case "ie":
        return new InternetExplorerDriver();
      case "safari":
        return new SafariDriver();
      case "headless":
        return new HTMLUnitDriver(); // Navegador Headless
    }
  }

}
```

Algoritmo 6. Creación de navegadores para pruebas funcionales.

4.2.3.2. Etapa de Construcción

Hasta el momento, que el código compile y pase la fase de pruebas unitarias indica una aceptación en esta etapa de V&V, pero esto no representa una aceptación en términos de características no funcionales [7].

En esta etapa, donde la aplicación no se encuentra en funcionamiento, no se pueden llevar a cabo pruebas de seguridad o capacidad, pero si es posible la ejecución de herramientas de análisis de código que provean retroalimentación respecto a características no funcionales del código como mantenibilidad, cobertura de pruebas y secciones del código con violaciones de

[53] https://github.com/SeleniumHQ/selenium/wiki/HtmlUnitDriver

seguridad. Si el código no cumple con los umbrales preestablecidos para estas métricas, el flujo de Integración Continua debe tenerse en esta etapa de V&V de la misma manera en que lo hace cuando una prueba unitaria falla.

En la sección 2.6.1, se detalla la importancia del análisis de código, como así también métricas útiles, como, por ejemplo:

- Cobertura de pruebas.
- Cantidad de código duplicado.
- Complejidad ciclomática.
- Acoplamiento aferente y eferente.
- Estilos de código.

Para este CTIL, el objetivo principal de esta etapa de V&V es la incorporación de herramientas de inspección de código o análisis estático del código fuente para aumentar la confiabilidad de esta etapa mediante la verificación de características no funcionales del código fuente.

Asimismo, antes de que esta etapa finalice con el despliegue de la aplicación a un ambiente de pruebas, el despliegue debe ser verificado para evitar fallos en la siguiente etapa. Esto se logra mediante la incorporación de pruebas de instalación o despliegue [164], [165].

PRÁCTICA 3.2.1: ANÁLISIS ESTÁTICO DEL CÓDIGO FUENTE

Las revisiones de código realizadas por otro/s desarrollador/es son efectivas para verificar la calidad del código fuente pero a su vez son subjetivas y requieren tiempo [9]. Cómo en esta etapa, la aplicación aún no se encuentra en ejecución, es posible utilizar herramientas de análisis que son configurables para detectar violaciones de código fuente de manera objetiva, con relación a características no funcionales y además mucho más rápidas de ejecutar ya que permiten automatizar este proceso. La Tabla 30 presenta la práctica de análisis estático del código fuente.

Tabla 30. Practica: Análisis estático del código fuente.

Análisis estático del código fuente	
Objetivo	Verificar características no funcionales del código como mantenibilidad, cobertura de pruebas y secciones del código con violaciones de seguridad.

Análisis estático del código fuente		
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Falta de cobertura en características no funcionales, como mantenibilidad y seguridad del código.	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Existe una herramienta que mide la cobertura de las pruebas unitarias.
Tareas	1. Selección e instalación de una herramienta de análisis estático del código fuente. 2. Configuración de los umbrales de calidad.	

Tarea 3.2.1.1: Selección e instalación de una herramienta de análisis estático del código fuente

En la actualidad, existen muchas herramientas de este tipo. En la sección 2.6.1 y en los artículos de Irrazábal [86] y Rodríguez et al. [89] se mencionan muchas de estas herramientas dependiendo de las necesidades y los requerimientos no funcionales que se desean medir en relación al código fuente.

Según la herramienta que se seleccione, el proceso de su instalación será distinto. Sin embargo, todas deben ser configuradas de tal modo que el análisis de código ocurra posterior a la compilación y ejecución de pruebas unitarias [7].

De todas las herramientas que existen en la actualidad, la más usada en ambientes Integración Continua o Entrega Continua, es SonarQube [166]–[170]. SonarQube admite más de 25 lenguajes de programación y es utilizada por más de 85,000 organizaciones.

Una de las sugerencias mencionadas durante la validación del modelo (apartado 6.1.4) fue la incorporación de reglas para la detección de vulnerabilidad de seguridad dentro del análisis estático del código fuente, también es posible la detección de vulnerabilidades de seguridad. Una herramienta utilizada para este fin es Fortify[54]. Es importante incorporar este tipo de validaciones en etapas temprana del flujo de Integración Continua, sobre todo en aplicaciones donde la seguridad es un atributo de calidad primordial.

[54] https://www.microfocus.com/es-es/products/static-code-analysis-sast/overview

Tarea 3.2.1.2: Configuración de los umbrales de calidad

Para aumentar las posibilidades de éxito utilizando una herramienta de análisis estático de código fuente, la misma debe estar ajustada adecuadamente para la estructura del código, la arquitectura y los requerimientos de negocio. Estas configuraciones permiten maximizar los defectos identificados mientras se minimizan los falsos positivos [9]. El valor del análisis estático reside en los conjuntos de reglas y umbrales que han sido seleccionados y como resultado, elegir los conjuntos correctos es imprescindible [84].

La mayoría de las herramientas permiten configurar reglas y umbrales. El conjunto de umbrales se denomina compuertas de calidad[55]. Para atravesarlas, el código debe cumplir de forma exitosa cada uno de los umbrales establecidos.

Antes de realizar los ajustes mencionados y según la necesidad de cada proyecto primero se deben responder a preguntas como: ¿cuál es la cobertura mínima de pruebas unitarias?, ¿qué cantidad de pruebas unitarias pueden no ser ejecutadas? ¿a qué grupo pueden pertenecer?, ¿qué tasa de fallo en pruebas unitarias es aceptable?, ¿qué directorios, paquetes o carpetas son excepciones al análisis?, ¿cuál es la cantidad mínima obligatoria de líneas de código no comentadas o documentadas?, ¿qué tan grave es no utilizar una determina práctica?, ¿qué prioridad tienen para el proyecto las complejidades de código?, ¿qué estilos deben ser implementados de forma obligatoria en el código?, etc.

Preguntas de este tipo permiten ajustar reglas, umbrales y compuertas de calidad. Además, es necesario contar previamente con una herramienta de cobertura de pruebas unitarias, ya que el análisis estático de código fuente requiere en su configuración, la ruta en donde se generan estos reportes de cobertura.

Cuando se ejecute el análisis, identificará si el código cumple con todos los umbrales de calidad que ha establecido; de lo contrario, fallará la puerta de calidad y por ende toda la etapa de V&V.

PRÁCTICA 3.2.2: PRUEBAS DE DESPLIEGUE E INSTALACIÓN

Con el objetivo de evitar que las pruebas funcionales automatizadas de la siguiente etapa fallen debido a que la aplicación no fue instalada o desplegada correctamente en un determinado ambiente, se utilizan pruebas de instalación o despliegue [164], [165]. Son pequeños chequeos

[55] https://docs.sonarqube.org/latest/user-guide/quality-gates/

que se realizan para verificar que la aplicación se encuentra operativa. Esta práctica se describe en la Tabla 31.

Tabla 31. Practica: Pruebas de despliegue e instalación.

Pruebas de despliegue e instalación		
Objetivo	Verificar si la instalación o despliegue de la aplicación en un entorno específico se realizó correctamente.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas que fallan debido a que la aplicación no fue instalada correctamente.	Pruebas no deterministas
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Se tienen identificados los servidores y servicios claves que indican si la aplicación está 100% en correcta ejecución.	Existe una herramienta para automatizar pruebas.
Tareas	1. Creación de pruebas de despliegue e instalación.	

Tarea 3.2.2.1: Creación de pruebas de despliegue e instalación

No se puede esperar que una aplicación funcione solo porque el proceso de despliegue concluyó; se debe verificar que la misma ha sido instalada o desplegada y configurada con éxito mediante un proceso de verificación automatizado [165]. Cada despliegue debe estar seguido de un conjunto de verificaciones de despliegue automatizadas. Estas verificaciones deben ser reutilizables, para que las mismas pruebas y comprobaciones sean usadas después de cada despliegue o instalación, sin importar el entorno.

Estas prueban deben verificar solo el despliegue y no las funcionalidades de la aplicación. En la primera versión del modelo, el objetivo de este tipo de pruebas eran verificar que el servidor principal donde la aplicación se ejecuta esté corriendo. Sin embargo, como parte de las mejoras introducidas luego del cuarto ciclo de validación (sección 6.2.5.4), estas pruebas deben centrarse en las interfaces entre sistemas (por ejemplo, direcciones IP o contrafuego), propiedades de configuración (por ejemplo, configuración de conexión de la base de datos) y signos básicos de vida del sistema (por ejemplo, si la aplicación responde) [165]. Es por ello que de antemano se

deben conocer cuáles son todas las configuraciones correctas, como así también los servidores y servicios que deben estar funcionando.

Las pruebas de instalación o despliegue generalmente se pueden automatizar utilizando la misma herramienta que se usa para las pruebas funcionales, como se muestra en el Algoritmo 7. Algunos servicios como Amazon, proveen mecanismos para ver si las instancias o servicios web se encuentran funcionando, como por ejemplo las verificaciones de salud[56].

```
public class DeploymentTest {

	public static final String BOOKING_HEALTH_CHECK =
				"http://my-app/api/v1/booking/healthcheck";
	public static final String PRINTING_HEALTH_CHECK =
				"http://my-app/api/v1/printing/healthcheck";
	public static final String INVENTARY_HEALTH_CHECK =
				"http://my-app/api/v1/inventary/healthcheck";
	public static final String USERS_HEALTH_CHECK =
				"http://my-app/api/v1/users/healthcheck";

	public static final String SQL_DB = "http://my-app/api:1433";
	public static final String MONGO_DB = "http://my-app/api:27017";

	public static final String WEB_CLIENT = "http://my-app";


	@Test // isServiceRunning() verifica que el código de estado sea 200
	public void verifyBackendRunning() {
		Assert.assertTrue(isServiceRunning(BOOKING_HEALTH_CHECK));
		Assert.assertTrue(isServiceRunning(PRINTING_HEALTH_CHECK));
		Assert.assertTrue(isServiceRunning(INVENTARY_HEALTH_CHECK));
		Assert.assertTrue(isServiceRunning(USERS_HEALTH_CHECK));
	}

	@Test // isDBUp() verifica que la DB se encuentre en funcionamiento
	public void verifyDBsRunning() {
		Assert.assertTrue(isDBUp(SQL_DB));
		Assert.assertTrue(isDBUp(MONGO_DB));
	}

	@Test // isAppRunning() contiene unos pequeños tests de Selenium
		// que verifican que los componentes principales de la
		// aplicación se muestren
	public void verifyDBsRunning() {
		Assert.assertTrue(isAppRunning(WEB_CLIENT));
	}

}
```

Algoritmo 7. Ejemplo de pruebas de instalación y despliegue.

[56] *Health Check* - https://aws.amazon.com/es/builders-library/implementing-health-checks/

4.2.3.3. Etapa de pruebas funcionales

Las pruebas funcionales son utilizadas como una fase de regresión para verificar que las funcionalidades más importantes del sistema. Sin embargo, hasta este momento, los escenarios alternativos y negativos, son llevados a cabo mediante los equipos de pruebas manuales, durante las pruebas exploratorias.

Para ampliar la cobertura de las pruebas automatizadas que se realizan en esta etapa de V&V, en este nivel CTIL se comienzan a automatizar las pruebas alternativas y negativas, como así también las pruebas que se documentaron para evitar defectos encontrados durante las pruebas de aceptación de usuario. Esto permite ampliar la cobertura de las pruebas automatizadas en esta etapa. También se incorpora la automatización de pruebas de extremo a extremo (E2E), a partir de las sugerencias de los expertos mencionadas en la sección 6.1.4 y 6.2.5.4.

Además, se introducen prácticas específicas para aumentar la confiabilidad. En primer lugar, recomendaciones para evitar los errores en sitios web o aplicaciones con contenido dinámico o llamadas asíncronas, mediante la implementación de funciones de esperas explícitas. Luego, se incorporan pruebas de despliegue e instalación, que verifica que la aplicación se encuentre lista antes de ejecutar las pruebas funcionales. Finalmente, se propone un mecanismo de omisión de pruebas para minimizar falsos positivos, junto con sistemas de re-ejecución de pruebas que fallan.

PRÁCTICA 3.3.1: AUTOMATIZACIÓN DE PRUEBAS ESPECIALES

La ejecución de pruebas exploratorias es un proceso repetitivo pero importante que debe realizarse en cada flujo de Integración Continua. Sin embargo, los escenarios negativos y alternativos de funcionales previamente desarrolladas pueden automatizarse para reducir los tiempos en pruebas exploratorias y aumentar la cobertura de las pruebas funcionales [165]. Asimismo, cuando un defecto es detectado y documentado, este escenario también debe ser automatizado para evitar la reaparición del mismo en posteriores integraciones de código [7]. La práctica de automatización de pruebas alternativas, negativas y de casos de pruebas diseñados a partir de defectos encontrados, se denomina automatización de pruebas especiales (

Tabla 32).

Tabla 32. Practica: Automatización de pruebas especiales.

Automatización de pruebas especiales		
Objetivo	Ampliar el alcance de las pruebas automatizadas, a las salidas de las pruebas de aceptación manuales, es decir, escenarios alternativos, negativos y pasos utilizados al detectarse un defecto para evitar su reaparición.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas exploratorias que demandan mucho tiempo.	Tiempos de ejecución elevados.
	Pruebas funcionales que solo detectan errores en partes críticas de la aplicación.	Poca cobertura de pruebas en Entrega Continua.
	Pruebas que no son ejecutadas correctamente de forma manual brindando falsos positivos.	Pruebas no deterministas.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Los escenarios alternativos y negativos están documentados.
		Los defectos encontrados durante el proceso de pruebas manuales están documentados.
		Existen pruebas funcionales automatizadas.
Tareas	1. Automatización de escenarios alternativos y negativos. 2. Automatización de pruebas sobre defectos.	

Tarea 3.3.1.1: Automatización de escenarios alternativos y negativos

La automatización de escenarios alternativos y negativos no debería ser una tarea compleja de implementar, puesto que son solo variaciones en las pruebas funcionales que ya se encuentran automatizadas.

Esto representa una oportunidad para que especialistas en pruebas manuales que no tengan mucho conocimiento de programación, colaboraren con las pruebas automatizadas con

la supervisión de desarrolladores o expertos en pruebas automatizadas. Esto también significa una ventaja para aquellos proyectos que no dispongan de tiempo para automatizar más pruebas y que no puedan incorporar más recursos.

Sin embargo, sea quien sea el encargado de automatizar estos escenarios, se debe contar con los casos de pruebas documentados para ellos, mínimamente con una breve descripción de los mismos.

Tarea 3.3.1.2: Automatización de pruebas sobre defectos

Unas de las maneras de asegurar que un defecto no vuelva a ocurrir, es mediante la automatización de los pasos que lo detectaron la primera vez. Esta tarea fue añadida luego de la validación del modelo, en el cuarto ciclo (apartado 6.2.5.4). Al igual que la automatización de escenarios alternativos y negativos, esta tarea puede ser llevada a cabo por miembros del equipo con muy poco conocimiento de programación, ya que, generalmente, también son variaciones de otros casos de pruebas ya automatizados.

La entrada para este proceso de automatización, son los defectos. Por esta razón, todo defecto debe ser documentado en la herramienta de seguimiento de incidentes durante la etapa de pruebas exploratorias.

PRÁCTICA 3.3.2: AUTOMATIZACIÓN DE PRUEBAS DE EXTREMO A EXTREMO

Si bien las pruebas deben ser atómicas [7] y los escenarios de extremo a extremo (E2E) pueden ser difíciles de mantener, la automatización de las mismas es la mejor manera de asegurar de que todo el sistema funcionará una vez que sea desplegado en un entorno [19]. Esta es una práctica del modelo y se presenta en la Tabla 33.

Tabla 33. Practica: Automatización de pruebas de extremo a extremo

Automatización de pruebas de extremo a extremo		
Objetivo	Automatizar pruebas que verifiquen si la aplicación funciona como está diseñada en todos los niveles y en todos los subsistemas.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas funcionales que solo detectan errores en un nivel del sistema.	Poca cobertura de pruebas en Entrega Continua.

Automatización de pruebas de extremo a extremo		
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Existen pruebas funcionales automatizadas.
Tareas	1. Automatización de pruebas de extremo a extremo horizontales. 2. Automatización de pruebas de extremo a extremo verticales.	

Tarea 3.3.2.1: Automatización de pruebas extremo a extremo horizontales

Las pruebas extremo a extremo horizontales, son aquellas que verifican flujos de negocio completos como, por ejemplo, el proceso de compra en una plataforma de comercio electrónico o la reserva de un vuelo en la página de una aerolínea. Este tipo de pruebas son más comunes que las verticales, ya que ocurren en una misma capa del sistema, como por ejemplo la interfaz gráfica de un sitio web [19]. A su vez, esto hace que su implementación pueda realizarse con la misma herramienta.

Una vez que se comiencen a automatizar flujos horizontales, estas pruebas deben pertenecer a un lote diferente del resto de las pruebas, ya que, en alguna ocasión, su ejecución puede ser necesaria de forma aislada del resto de las pruebas que son atómicas [19].

Tarea 3.3.2.2: Automatización de pruebas extremo a extremo verticales

Una prueba de extremo a extremo vertical es una prueba que verifica el comportamiento del sistema, en todas las capas al realizar una acción en una de ellas. Existen diferentes combinaciones de flujos, donde se destacan:

- Acción en la GUI – Verificación en la BD o sistema de archivos.
- Acción en una de las API – Verificación en la BD o sistema de archivos.
- Acción en la GUI – Verificación en otra aplicación interna o externa.
- Acción en una de las API – Verificación en otra aplicación interna o externa.

La automatización de este tipo de pruebas es una tarea muy complicada, ya que requiere de conocimientos en varias tecnologías para permitir la interacción con las diferentes capas del sistema o subsistemas.

PRÁCTICA 3.3.3: GESTIÓN DE ASINCRONÍA Y TIEMPOS DE ESPERA

Para las pruebas funcionales, dependiendo de la naturaleza de la aplicación, la asincronía puede ser imposible de evitar. Este problema puede ocurrir no solo con sistemas explícitamente asíncronos, sino también con cualquier sistema que use hilos o transacciones. En dichos sistemas, es posible que la llamada que realice deba esperar a que se complete otro subproceso o transacción. El problema aquí se reduce a la siguiente pregunta: ¿Ha fallado realmente la prueba o simplemente se está esperando que lleguen los resultados? Se ha descubierto que la estrategia más efectiva es construir mecanismos que aíslen la prueba de este problema [7]. En lo que respecta a la prueba en sí, se busca hacer que la secuencia de eventos que incorpora la prueba parezca síncrona. Esto se logra aislando la asincronía detrás de las llamadas síncronas. La práctica que permite gestionar los tiempos de espera para este tipo de problemas se denomina gestión de asincronía y tiempos de espera y se describe en la Tabla 34.

Tabla 34. Practica: Gestión de asincronía y tiempos de espera.

Gestión de asincronía y tiempos de espera		
Objetivos	Hacer que la secuencia de eventos que incorporan las pruebas parezca síncrona, mediante el aislamiento de la asincronía detrás de las llamadas síncronas.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Fallos producidos por resultados que demoran en aparecer.	Pruebas no deterministas
	Manejo de esperas para llamadas asincrónicas.	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	La aplicación es asincrónica o utiliza varios procesos o transacciones que se producen en paralelo.	Existen pruebas funcionales automatizadas.
Tareas	1. Identificar las pruebas que fallan debido a llamadas asíncronas. 2. Implementar esperas explícitas.	

Tarea 3.3.3.1: Identificación de las pruebas que fallan debido a llamadas asíncronas

El objetivo de esta tarea consiste en detectar las pruebas que fallan debido a llamadas asíncrona, componentes o elementos que aún no están cargados o renderizados en una página,

o simplemente porque el resultado de una acción no se completó. Esta tarea se realiza en primer lugar mediante el análisis de los reportes de ejecución generados por el marco de trabajo de pruebas, utilizando los mensajes de error, o las mismas excepciones.

Para validar que realmente son errores producidos por falta de tiempo, se pueden añadir esperas implícitas antes de realizar la acción que genera el error. Un ejemplo de espera implícita puede verse en el siguiente algoritmo (ver Algoritmo 8).

```
private boolean isTicketGenerated() {

      Wait(DELAY_PERIOD);

      return getTicket().isDisplayed();

}
```

Algoritmo 8. Espera implícita.

En el Algoritmo 8, antes de verificar que el comprobante se muestre, se realiza una espera implícita por un tiempo DELAY_PERIOD, antes de fallar. Si la variable DELAY_PERIOD es una constante con un tiempo relativamente largo y el error que ocurría antes era por falta de tiempo, entonces la prueba automatizada dejará de fallar.

Sin embargo, el inconveniente de este enfoque es que estas variables DELAY_PERIOD suman mucho tiempo de ejecución. La próxima tarea resuelve este problema.

Tarea 3.3.3.2: Implementación de esperas explícitas

Una vez que las pruebas que fallan por tiempos de esperas agotados fueron detectadas y solucionados utilizando el enfoque anterior, el siguiente paso es optimizar las esperas para ahorrar el tiempo de ejecución total de las pruebas automatizadas. Para ello, se utiliza un enfoque de espera explícita, es decir, en lugar de esperar a que se cumpla el período más largo aceptable (DELAY_PERIOD), se implementan pequeñas esperas con reintentos. Esto puede verse en el Algoritmo 9.

```
private boolean isTicketGenerated() {
      TimeStamp testStart = TimeStamp.NOW;
      boolean ticketFound = false;

      do {
            if (getTicket().isDisplayed()) return true;
            Wait(SMALL_PAUSE);
      } while ( TimeStamp.NOW < testStart + DELAY_PERIOD );
      return false;

}
```

Algoritmo 9. Espera explícita.

En el Algoritmo 9, en lugar de esperar el máximo tiempo de espera, se realiza simplemente una pequeña pausa. Esta prueba con este enfoque es mucho más eficiente que la versión anterior de espera implícita, siempre que SMALL_PAUSE sea pequeña en comparación con DELAY_PERIOD (generalmente dos o más órdenes de magnitud más pequeños), ya que si DELAYED_PERIOD fuera 5 segundos, SMALL_PAUSE 1 segundo y el ticket se generó en 3 segundos, entonces se ahorrarían 2 segundos.

PRÁCTICA 3.3.4: ESTRATEGIA DE OMISIÓN DE PRUEBAS

Un caso de prueba no es solo la verificación de una condición, sino un conjunto de pasos, precondiciones y datos de pruebas. En las pruebas automatizadas, cuando no se dan las precondiciones, no existen los datos de pruebas o no se pueden ejecutar unos de los pasos previos a la verificación de la condición, se producen fallos. Sin embargo, estos fallos no significan que la condición no se haya cumplido, sino falsos positivos. Si se verifican que las precondiciones se den, los datos de pruebas existan y los pasos previos a las condiciones se ejecuten correctamente, es posible omitir las pruebas antes de que produzcan falsos positivos. Este práctica se denomina omisión de pruebas [20] y se presenta en la Tabla 35.

Tabla 35. Practica: Estrategia de omisión de pruebas.

Estrategia de omisión de pruebas		
Objetivos	Omitir la ejecución de pruebas que fallarán antes que se verifique la condición principal.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Fallos no relacionados con las condiciones que verifican las pruebas.	Pruebas no deterministas

Estrategia de omisión de pruebas		
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Existen pruebas funcionales automatizadas.
Tareas	1. Identificación de las causas comunes de errores en la ejecución de los pasos, por falta de datos de pruebas y no cumplimiento de precondiciones. 2. Definición de la estrategia de omisión de pruebas. 3. Refactorización de las pruebas automatizadas con la estrategia de omisión de pruebas.	

Tarea 3.3.4.1: Identificación de las causas comunes de errores en la ejecución de los pasos, por falta de datos de pruebas y no cumplimiento de precondiciones

Otra de las causas comunes de pruebas no deterministas, son los errores que se producen en los pasos previos a la verificación de la condición que está evaluando la prueba. Esto también abarca al no cumplimiento de una precondición de la prueba, como así también la falta de datos de pruebas necesarios para la ejecución [20]. Además, dependiendo del tipo de proyecto, la plataforma, la cantidad de sistemas y subsistemas involucrados, existen otras causas que producen errores antes de llegar a la verificación final de la prueba, tal y como se menciona a lo largo del capítulo 3. El objetivo de esta tarea es la determinación de todas estas causas.

En primer lugar, en todo proyecto, la mayoría de los casos de pruebas tienen precondiciones, así sean tan simples como navegar a una determinada URL. De este modo, el no cumplimiento de precondiciones debe ser el primer tipo de causa.

En segundo lugar, si la aplicación utiliza datos de pruebas, la falta de estos genera pruebas no deterministas y en consecuencia falsos positivos [20]. Esto convierte a la falta de datos de pruebas en el segundo tipo de causa.

Finalmente, los errores en la ejecución de los pasos previos a la verificación de las condiciones se pueden dar por diversos causas [11], [14], [123]: inestabilidad del ambiente, cambios en los elementos de la GUI (botones, ventanas, campos, etc.), sistemas externos o de terceros que no están disponibles, problemas de red, entre otros. Utilizando los reportes de ejecución de las pruebas automatizadas, es posible identificar todas estas causas e ir sumándolas a la lista.

No es necesario la finalización de esta tarea para el comienzo de la siguiente, ya que en la medida que se sigan ejecutando las pruebas, se seguirán identificando nuevas causas. Lo que

hace que la identificación de causas, la estrategia de omisión y refactorización de las pruebas sea un proceso iterativo.

Tarea 3.3.4.2: Definición de la estrategia de omisión de pruebas.

Una vez identificada las causas de falsos positivos más frecuentes no relacionadas a la verificación principal de la prueba, se define la estrategia de omisión de pruebas. Cada causa debe tener su mecanismo de omisión y esto se debe documentar junto con el resto de los procedimientos del proyecto [11].

El mecanismo de omisión consiste en verificar que se cumpla un estado antes de pasar al siguiente. En caso de que el estado no se cumpla se generara una retroalimentación indicando que la prueba no puede continuar su ejecución.

A continuación, se muestran algunos ejemplos de la estrategia de omisión de pruebas de un proyecto web, compuesto por una interfaz gráfica y microservicios:

- Verificar que, al navegar a una página, la misma haya cargado correctamente antes de la ejecución del próximo paso. En caso de que no haya cargado, omitir la prueba.
- Verificar que las credenciales del usuario a utilizar existan y sean correctas, cuando corresponda, antes de la ejecución de las pruebas. En caso de que no existan o sean incorrectas, omitir la prueba.
- Capturar todos los errores producidos por tiempos de espera agotados y omitir la prueba.
- Verificar que todas las llamadas a los microservicios sean correctas, antes de hacer validaciones sobre los mismos. En caso de que las respuestas no sean correctas, omitir la prueba.
- En los escenarios en donde el usuario deba tener la sesión iniciada, validar que esto haya ocurrido antes de comenzar con la ejecución de los pasos de la prueba. En caso de que el usuario no haya iniciado sesión correctamente, omitir la prueba.
- Capturar todos los errores producidos por el driver de GUI, cuando no puede interactuar con algún elemento gráfico y omitir la prueba.

Una vez que la estrategia esté creada, la misma debe ser publicada y compartida con los integrantes del proyecto, del equipo o al menos con los involucrados en la automatización y mantenimiento de las pruebas funcionales.

Tarea 3.3.4.3: Refactorización de las pruebas automatizadas con la estrategia de omisión de pruebas

Esta tarea (añadida como sugerencia luego de la validación del modelo en la sección 6.2.5.4) consiste en la refactorización de las pruebas automatizadas utilizando la estrategia de omisión de pruebas creada previamente. El criterio a utilizar para seleccionar las pruebas que serán refactorizadas al principio debe ser elegido teniendo en cuenta las que están produciendo falsos positivos. Luego, se continúan con las que no generan errores, teniendo en cuenta que las mismas pueden hacerlo en el futuro.

La implementación del mecanismo de omisión dependerá del marco de trabajo de pruebas. La mayoría de estas herramientas, tienen un sistema que permite omitir la prueba utilizando instrucciones especiales. En el Algoritmo 10 se muestra un ejemplo de omisión de pruebas utilizando Java y C#, con diferentes marcos de trabajo de pruebas xUnit.

```
// Java - JUnit / TestNG
@Test
public void verifyOneMessage() {
      if (!page.isLoaded())
            throw new SkipException("The page has never loaded");
      if (!user.isLoggedIn())
            throw new SkipException("The user is not logged in");

      // keep running the test
}

/* C# - NUnit */
@Test
public void VerifyOneMessage()
{
      if (!_page.IsLoaded())
            Assert.Ignore("The page has never loaded");
      if (!_user.IsLoggedIn())
            Assert.Ignore("The user is not logged in");

      /* keep running the test */
}
```

Algoritmo 10. Omisión de pruebas en Java y C#.

PRÁCTICA 3.3.5: SISTEMA DE RE-EJECUCIÓN DE PRUEBAS FALLIDAS

Cuando una prueba automatizada falla, existe la posibilidad de que ese resultado sea un falso positivo. Uno de los más conocidos y utilizados mecanismos de identificación de falsos

positivos, es la re-ejecución de pruebas que son fallidas [171], [172]. Si después de re-ejecutar la prueba, la misma no falla, entonces la prueba se considera como “no determinista” y se puede evaluar si ejecutar una tercera vez para determinar su resultado o simplemente quitarla de los resultados finales y luego refactorizarla. Esta práctica se describe en la Tabla 36.

Tabla 36. Practica: Sistema de re-ejecución de pruebas fallidas.

Sistema de re-ejecución de pruebas fallidas		
Objetivos	Re ejecutar los casos de pruebas automatizados antes de considerarlo como fallidos.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Fallos no relacionados con las condiciones que verifican las pruebas.	Pruebas no deterministas
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Existen pruebas funcionales automatizadas.
Tareas	1. Creación del sistema de re-ejecución de pruebas.	

Tarea 3.3.5.1: Creación del sistema de re-ejecución de pruebas

La creación de un sistema de re-ejecución de pruebas varía mucho dependiendo del marco de trabajo de pruebas. En algunos con lenguajes orientados a objetos, se deben implementar interfaces especiales conocidas como interfaces de reintento, mientras que en otros lenguajes, se utilizan herramientas externas, dependencias o librerías externas.

Independientemente de la implementación del sistema de re-ejecución de pruebas, lo que debe determinarse es el criterio de aceptación para las pruebas que van a ser re-ejecutadas. Si bien esto depende mucho del tipo de aplicación que se está desarrollando, no basta con re-ejecutar solo una vez más la prueba cuando falla. Si una prueba falla y luego de re-ejecutarla arroja un resultado exitoso, entonces es necesario re-ejecutar como mínimo una vez más, para determinar el resultado final [172].

Además, cualquier prueba que luego de re-ejecutarla arroje un resultado distinto del de la primera ejecución, debe ser marcada como “inestable” o agregada en una “lista de cuarentena” [172], para su posterior refactorización. Los equipos deben seguir un proceso estricto al detectar una prueba no determinista. Después de registrarlas y quitarlas de los lotes de pruebas de

ejecución principales, se deben investigar las causas de su no determinismo. Separarlas del resto de las pruebas es importante para mantener la confiabilidad de las demás.

PRÁCTICA 3.3.6: EJECUCIÓN DE PRUEBAS FUNCIONALES PROGRAMADAS

Esta práctica consiste en la ejecución de pruebas funcionales en el servidor de Integración Continua con frecuencia en diferentes momentos del día permite revelar pruebas no deterministas. La misma se presenta a continuación en la Tabla 37.

Tabla 37. Práctica: Ejecución de pruebas funcionales programadas.

Ejecución de pruebas funcionales programadas		
Objetivos	Ejecutar las pruebas funcionales de forma programada en el servidor de Integración Continua, para detectar pruebas no deterministas.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Fallos no relacionados con las condiciones que verifican las pruebas.	Pruebas no deterministas
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	El servidor de Integración Continua permite programación de tareas.	Existen pruebas funcionales automatizadas.
Tareas	1. Programar la ejecución de pruebas funcionales.	

Tarea 3.3.6.1: Programar la ejecución de pruebas funcionales

Hasta el momento, las pruebas funcionales se ejecutan como parte de esta etapa, en el conducto de despliegue y en la PC del desarrollador. Sin embargo, con el objetivo de aumentar la detección de pruebas no deterministas, las pruebas funcionales se pueden ejecutar con mayor frecuencia en el servidor de Integración Continua y no solo cuando se envían cambios al repositorio de control de versiones.

Para ello, la mayoría de los servidores de Integración Continua permiten la programación de una tarea en un horario o momento del día determinado [9].

Lo ideal es programar la ejecución de las pruebas funcionales, en los momentos en que haya menos utilización del servidor, como por ejemplo por la noche.

Finalmente, la re-ejecución de pruebas también debe estar habilitada en esta tarea programada, ya que este sistema es el primero en detectar las pruebas no deterministas, previo a una revisión manual de los reportes para confirmar los verdaderos fallos y los falsos positivos.

PRÁCTICA 3.3.7: SISTEMA DE GENERACIÓN Y ELIMINACIÓN DE DATOS DE PRUEBAS

Otra de las causas de las pruebas no deterministas, son los datos corruptos o inexistentes. Esto se debe principalmente a que los datos se encuentran almacenados en bases de datos o archivos externos. Una de las soluciones a este problema, es utilizar la práctica de pruebas unitarias de preparar los datos antes de la ejecución de las pruebas y eliminarlos cuando la misma finalice. Esta práctica se conoce como sistema de generación y eliminación de datos de pruebas [20] y se describe en la Tabla 38.

Tabla 38. Práctica: Sistema de generación y eliminación de datos de pruebas.

Sistema de generación y eliminación de datos de pruebas		
Objetivos	Crear los datos de pruebas únicamente antes de usarlos y eliminarlos después de su uso.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Fallos relacionados a datos corruptos o faltantes.	Pruebas no deterministas
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Se tiene acceso de lectura y escritura al medio donde se almacenan los datos.	Existen pruebas funcionales automatizadas.
Tareas	1. Crear funciones de creación y eliminación de datos de pruebas. 2. Implementar las funciones creadas.	

Tarea 3.3.7.1: Codificar funciones de creación y eliminación de datos de pruebas

Si se tiene acceso de lectura y escritura al medio donde se almacenan los datos (base de datos, archivo, etc.) la función de creación de datos de pruebas consiste en un script que invoque un servicio web o ejecute una consulta a una base de datos o instrucción para manipular un archivo. Este script recibe como parámetros los datos de pruebas a insertar y el resultado final, es la actualización del medio de persistencia. En el Algoritmo 11 se muestra un ejemplo de creación de un usuario de prueba utilizando tanto una API como una consulta SQL.

Por otro lado, utilizando el mismo criterio y las mismas sentencias, se crea la función de eliminación de datos de pruebas. Por ejemplo, utilizando un verbo HTTP “Delete” o un “Post” con un parámetro de borrado, o con “DELETE” si fuera una instrucción SQL.

Tarea 3.3.7.2: Implementar las funciones creadas

Como se ha mencionado en otras prácticas, la mayoría de los marcos de trabajo de pruebas automatizadas soportan métodos especiales que se ejecutan antes de las pruebas y después de ellas. De este modo, el objetivo de esta tarea es utilizar dichos métodos para invocar a las funciones generadas en la tarea anterior, como se muestra en el Algoritmo 12.

```
public class TestDataGenerator {

      public static final USERS_SERVICE = "http://api-environment/users";

      public boolean createUserAPI(String username, String password,
            String type) {
            HTTPClient client = new RestClient();
            Username newUser = new Username(username, password, type);
            HTTPResponse response = client.post(USERS_SERVICE, newUser);
            return response.getStatusCode == 200;
      }

      public boolean createUserInDB(String username, String password,
            String type) {
            try {
                  Connection con = DriverManager.getConnection(Url);
                  PreparedStatement stmt = con
                        .prepareStatement("insert into users (username,
                        password, type) values (?, ?, ?)");
                  stmt.setString(1, username);
                  stmt.setString(2, password);
                  stmt.setString(3, type);
                  stmt.executeUpdate();
                  con.close();
                  return true;
            catch (Exception e) {
                  return false;
            }
      }
}
```

Algoritmo 11. Script de generación de datos de pruebas.

```
public void TestExample {

	@Inject
	private TestDataGenerator generator;

	@BeforeClass
	public void setUp() {
		generator.createUserAPI("fullUser", "Pass1234$", "Full");
	}

	@AfterClass
	public void tearDown() {
		generator.deleteUserAPI("fullUser");
	}

	@Test
	public void test() { Test Steps...}

}
```

Algoritmo 12. Sistema de creación y eliminación de datos de pruebas.

PRÁCTICA 3.3.8: GENERACIÓN DE REPORTES PRECISOS

Otro de los problemas que atenta contra la confiabilidad de la etapa de pruebas, es la ambigüedad de los reportes de ejecución de las mismas (sección 3.5). Para evitarlo, se utilizan una serie de recomendaciones que logran un reporte preciso con resultados fáciles de comprender. La práctica que sugiere estas recomendaciones se presenta en la Tabla 39.

Tabla 39. Práctica: Generación de reportes precisos

Generación de reportes precisos		
Objetivos	Evitar que los reportes de resultados de las pruebas sean ambiguos.	
Descripción	Otro de los problemas que atenta contra la confiabilidad de la etapa de pruebas, es la ambigüedad de los reportes de ejecución de las mismas. Para evitarlo, se utilizan una serie recomendaciones que logran un reporte preciso con resultados fáciles de comprender.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Los resultados de las pruebas funcionales no se comprenden (Reportes ambiguos)	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**

Generación de reportes precisos		
	No aplica.	Existen pruebas funcionales automatizadas.
Tareas	1. Manipulación de Excepciones. 2. Implementación de un generador de reportes personalizado.	

Tarea 3.3.8.1: Manipulación de Excepciones

Uno de los problemas más comunes que existen a la hora de interpretar los reportes de ejecución de pruebas, son los mensajes técnicos producidos por excepciones [11]. Por ejemplo, si se estuviera utilizara la herramienta Selenium Webdriver, un error bastante frecuente es el producido cuando un elemento no existe en la página. Esto produce una excepción llamada “NoSuchElementException”. Para el desarrollador o el especialista en pruebas automatizadas es sencillo reconocer este error, pero para uno de los interesados en el sistema o personas no técnicas, no lo es.

Una manera de evitar reportes que contengan este tipo de excepciones técnicas, es mediante la manipulación de los mensajes. En el Algoritmo 13 se muestra un ejemplo de manipulación de errores de Selenium Webdriver.

```
public String refactorExceptionMessages(String exceptionMessage,
      String originalException) {

      switch (originalException) {
            case "NoSuchElementException":
                  return "A component is not available on the page.";
            case "ElementNotSelectableException":
                  return "The component cannot be selected or is not
                  available to interact.";
            case "ExpectedCondition":
                  return "The presence of a component was expected. It is
                  not available or was not fully loaded on the page.";
            case "TimeoutException":
                  return "Failed by Timeout";
            case "WebDriverException":
                  return "Unable to initialize WebDriver.";
            case "StaleElementReferenceException":
                  return "The component is no longer attached to the
                  page.";
            case "ElementClickInterceptedException":
                  return "The component is not clickable or cannot be
                  clicked.";
            default:
                  return exceptionMessage;
            }
      }
}
```

Algoritmo 13. Manipulación de excepciones.

Tarea 3.3.8.2: Implementación de un generador de reportes personalizado

El objetivo de esta tarea es la creación de una clase o script que permita generar los reportes con un formato personalizado. Esto brinda al desarrollador la posibilidad de incluir en el mismo capturas de pantalla, gifs, URLs donde se obtuvo el error, datos de pruebas, información de una petición, etc. Cuanta más información contenga el reporte, más fácil será interpretarlo. En la Fig. 50 se muestra un ejemplo de un reporte conteniendo toda esta información.

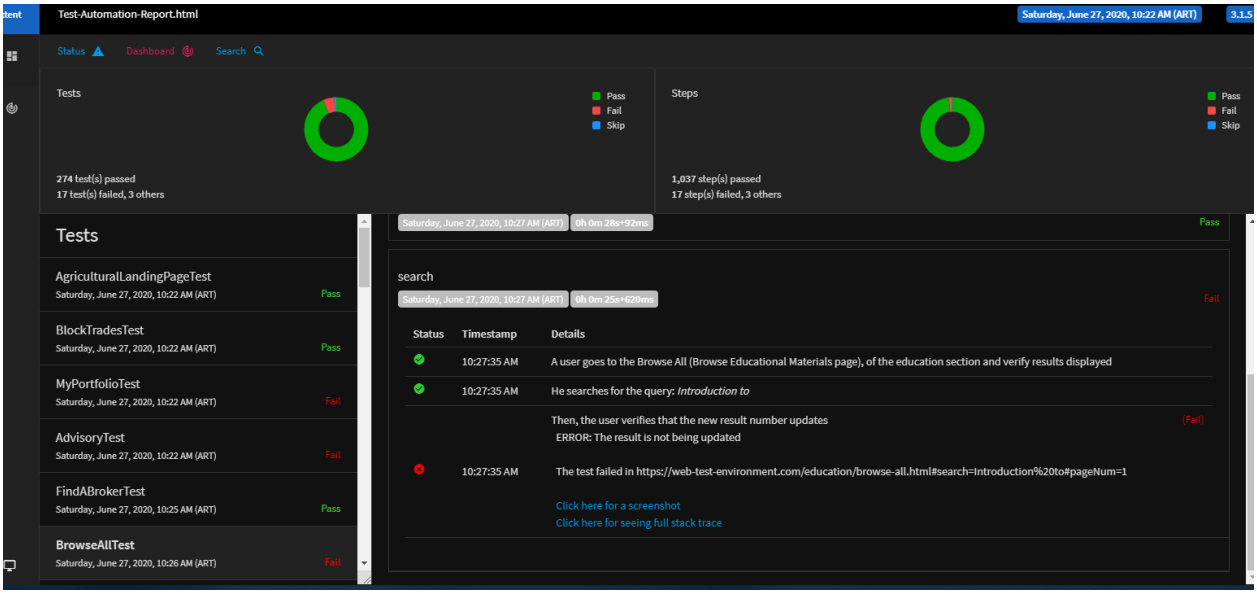


Fig. 50. Reporte de pruebas funcionales personalizado.

4.2.3.4. Etapa de pruebas no funcionales

Como se ha mencionado anteriormente, en este nivel CTIL el objetivo es aumentar la confiabilidad, por ello, en esta etapa de V&V se añaden más tipos de pruebas no funcionales, pertenecientes a las pruebas de seguridad, recomendadas por los expertos durante la validación del modelo (sección 6.1.4).

Las pruebas de seguridad son un conjunto de verificaciones que descubren vulnerabilidades, amenazas y riesgos en el software, aplicación o sistema que se está desarrollando, para prevenir ataques maliciosos. El propósito de las mismas es identificar todas las posibles debilidades del sistema que pueden resultar en problemas como pérdida de información, virus que destruyen el sistema, transacciones no deseadas, entre otras.

PRÁCTICA 3.4.1: GESTIÓN DE PRUEBAS DE SEGURIDAD

Las pruebas de seguridad consisten en la verificación de requisitos de seguridad del software relacionados a la confidencialidad, integridad, disponibilidad, autenticación, autorización y no reputación [173]. También incluye las pruebas para validar la capacidad del software de resistir a ataques [174]. Existen muchos tipos de pruebas de seguridad y en este nivel se abarcan las pruebas de penetración y análisis dinámico de vulnerabilidades. La práctica para gestionar este tipo de pruebas se presenta en la Tabla 40.

Tabla 40. Práctica: Gestión de pruebas de seguridad

Gestión de pruebas de seguridad		
Objetivos	Identificar las amenazas en el sistema y medir sus vulnerabilidades potenciales, para que el mismo no deje de funcionar o sea explotado.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Baja confiabilidad del software en términos de seguridad.	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	No aplica.
Tareas	1. Implementación de pruebas de seguridad.	

Tarea 3.4.1.1: Implementación de pruebas de seguridad

Existen diferentes tipos de pruebas de seguridad, pero su uso depende de la etapa de V&V en el conducto de despliegue o de la fase en el ciclo de desarrollo de software. En la Fig. 51 se muestran ejemplos de tipos de pruebas de seguridad en los procesos de desarrollo tradicionales.

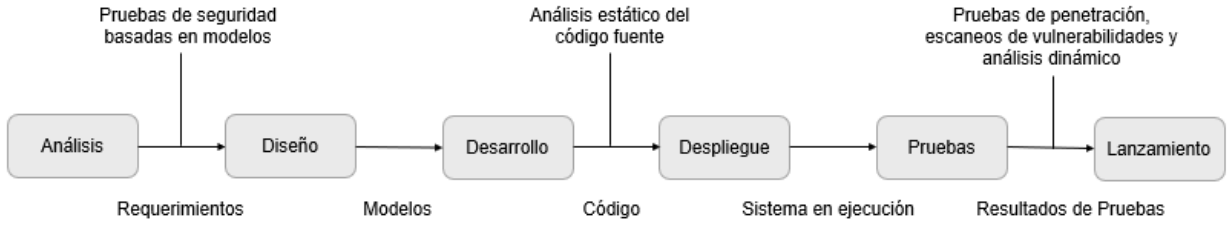


Fig. 51. Pruebas de seguridad en el ciclo de vida de desarrollo de software.

En los entornos de desarrollo continuo y utilizando las metodologías ágiles, las diferentes validaciones de seguridad deben realizarse cada vez que se dispara el flujo de Integración Continua. En este nivel CTIL, en las etapas tempranas del Conducto de Despliegue, se ha recomendado la incorporación de herramientas de verificación de vulnerabilidad de seguridad en el código como Fortify. Sin embargo, en relación con la Fig. 51 para los procesos tradicionales, en esta etapa de V&V se deben incorporar más pruebas de seguridad para aumentar la confiabilidad. Entre estas, se encuentran las pruebas de penetración y análisis dinámicos de vulnerabilidades, que se realizan sobre el sistema en ejecución, ya sea en un entorno de prueba o de producción [175]. Las pruebas de penetración buscan irrumpir en el software en ejecución,

pero desde un punto de vista ético, definiendo un requerimiento antes de ejecutar la prueba [174]. Las pruebas de penetración también se utilizan para probar la política y estándares de seguridad de una organización, la conciencia de seguridad de sus empleados y la capacidad de la organización para identificar y responder a incidentes de seguridad.

Existen estrategias y herramientas tanto comerciales como de código abierto y gratuitas[57], que permiten realizar pruebas de penetración y análisis de vulnerabilidades.

La elección de una o varias de estas herramientas dependerá de la necesidad de seguridad del proyecto. Una vez que la estrategia y las herramientas han sido elegidas, es importante incorporar la ejecución de la misma cada vez que se dispare el conducto de despliegue.

4.2.3.5. Etapa de pruebas de aceptación de usuario

Una de las principales diferencias que existe entre las pruebas manuales y las pruebas manuales ágiles, es que el último no verifica solamente el cumplimiento de los requerimientos, sino que busca que el software cumpla la necesidad del usuario a quién va dirigido [79]. En este sentido, también es importante verificar que el software sea fácil de usar y entender por los usuarios o el público objetivo, aun cuando estos no sean requisitos explícitos. Este grupo de verificaciones se conocen como pruebas de usabilidad y accesibilidad.

En esta etapa de V&V, como parte de los resultados de la validación del modelo (sección 6.1.4) y en línea con todo este CTIL de confiabilidad, el objetivo es incluir estas pruebas que permitan asegurar que el público objetivo podrá comprenden y utilizar con facilidad la aplicación en desarrollo.

PRÁCTICA 3.5.1: PRUEBAS DE USABILIDAD

Según las características de los usuarios finales del sistema, se requerirán pruebas de usabilidad desde la perspectiva del usuario. Las mismas se describen como parte de esta práctica (

Tabla 41).

[57] Ejemplos de estas herramientas son: W3af, Wireshark, John the Ripper, Wapiti, Wfuzz, SQLMap, Zed Attack Proxy

Tabla 41. Práctica: pruebas de usabilidad

Pruebas de usabilidad		
Objetivos	Lograr que el software sea fácil de usar para los usuarios finales del mismo.	
Descripción	Según las características de los usuarios finales del sistema, se requerirán pruebas de usabilidad desde la perspectiva del usuario.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Falta de usabilidad del software.	Poca cobertura de pruebas en Entrega Continua.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	El software es utilizado por una gran variedad de usuarios.	No aplica.
Tareas	1. Identificación de los diferentes usuarios de la aplicación. 2. Implementación de pruebas de usabilidad.	

Tarea 3.5.1.1: Identificación de los diferentes usuarios de la aplicación

Antes de realizar pruebas que verifiquen la usabilidad de la aplicación, es necesario reconocer a qué usuarios va destinado el software que se está desarrollando, es decir, quién es el público objetivo. Esta pregunta es sencilla de responder para aquellas organizaciones que cuenten con un equipo de marketing, o que cuenten con herramientas de análisis de mercado. Sin embargo, en todos los casos, quien solicita el producto será capaz de brindar esta información.

En el libro de Crispin y Gregory [79], se muestra un ejemplo de tipos de usuarios para una plataforma de comercio electrónico:

- Nancy Nueva, una ciudadana adulta que es nueva en el ámbito de las compras por internet y está nerviosa por miedo a que roben sus datos personales y tarjetas de crédito.
- Hudson Hacker, quien busca maneras de engañar al sistema en la página de pago.
- Enrico Ejecutivo, quien hace compras masivas por internet y envía regalos a todos sus clientes en todo el mundo.
- Betty Oferta, quien constantemente está en busca de los mejores precios.

- Debbie Indecisa, a quien le lleva mucho tiempo decidir qué cosa ella realmente quiere comprar.

Tarea 3.5.1.2: Implementación de pruebas de usabilidad

Partiendo de la tarea anterior, es posible ejecutar pruebas exploratorias desde el punto de vista de cada uno de los usuarios identificados. Este tipo de estrategia se conoce como “pruebas persona”.

Otro manera de ejecutar las pruebas persona, es tomar un personaje ficticio o una celebridad famosa e imaginar como ellos usarían la aplicación [79]. Por ejemplo, ¿podrá la Reina de Inglaterra navegar a través de todo el proceso de compra del sistema?, ¿cómo podría Homero Simpson buscar el producto que desea en el sitio? También se pueden considerar roles de los diferentes usuarios como “novato”, “intermedio” y “experto”.

Las “pruebas de navegación” constituyen otro aspecto de pruebas de usabilidad. Es importante verificar todos los enlaces de un sitio web, o que el orden de las pestañas/botones tenga sentido. Este tipo de verificaciones puede generar más casos de pruebas automatizables para la etapa de pruebas funcionales.

4.2.3.6. Análisis del nivel de mejora 3 y los problemas en Pruebas Continuas

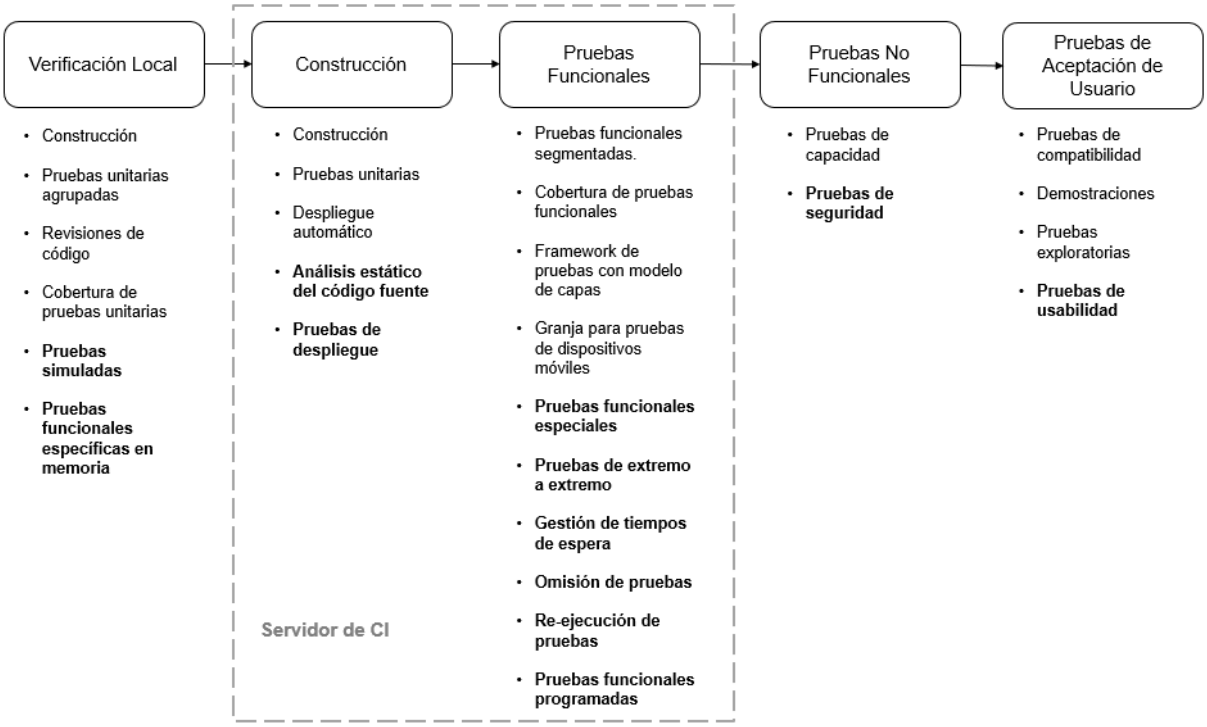


Fig. 52. Conducto de Pruebas Continuas en el nivel de mejora 3.

En el nivel de mejora 3 se definen estándares y patrones para el proceso de pruebas, y, en consecuencia, la etapa de ejecución de pruebas genera resultados confiables (ver Fig. 52). Esto significa que el problema que más se resuelve en este nivel, es el de las pruebas no deterministas (7 problemas de 9). Esto representa un 77,78%, que sumado al porcentaje obtenido en el nivel 2 (22,22%) suma un 100% (ver Fig. 53).

Asimismo, como se menciona anteriormente, la confiabilidad no solo significa pruebas deterministas, sino también mayor cobertura en su ejecución. Esto implica una mejora del 36% en los problemas de poca cobertura de pruebas en Entrega Continua (9 de 25). En total, con el grado de solución adquirido en los niveles 1 (36%) y 2 (28%), este problema es solucionado también en su totalidad (100%).

Finalmente, se percibe una mínima mejora en los tiempos de ejecución elevados, ya que una de las prácticas implementadas consiste en la automatización de casos de pruebas especiales (escenarios negativos y alternativos que se ejecutan en las pruebas exploratorias manuales). Esto genera un 5,56% (1 problema de 18), alcanzando un total de 38,89%, donde el nivel 1 generó un 11,11% y el nivel 2 un 22,22%.

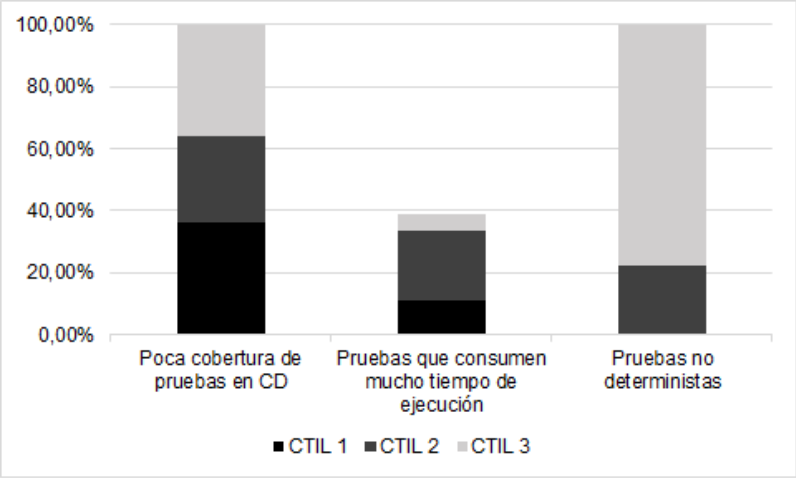


Fig. 53. Grado de resolución de los problemas de Pruebas Continuas al implementar el nivel 3 del modelo.

4.2.4. Nivel de Mejora 4

Hasta el momento, todas las prácticas incluidas en cada CTIL anterior, permitieron la construcción de un conducto de despliegue compuesto por diferentes etapas de V&V, donde cada una de ellas contiene diferentes tipos de pruebas estructuradas. Además, cualquier cambio en el código fuente de la aplicación que atraviese por todas estas etapas del conducto de despliegue, tendrá los niveles de calidad adecuados que permiten lanzarlo a producción. Esto se debe a que los resultados que arrojan las pruebas son confiables y también porque las mismas cubren las partes más importantes de la aplicación, y, por ende, disminuyen los riesgos de introducir errores graves en producción.

Sin embargo, todas estas prácticas que se fueron incorporando, sobre todo las que aumentan la cobertura y confiabilidad, introducen más tiempo en la ejecución de pruebas.

El nivel CTIL 4 se denomina **continuidad**. El mismo tiene como objetivo reducir el tiempo de ejecución de las pruebas y aumentar su eficiencia. Para ello, se proponen prácticas como selección de pruebas, paralelización, uso de API para ejecutar precondiciones, rotación de navegadores, entre otras. Esta etapa no solo abarca la implementación de prácticas para aprovechar mejor el tiempo en la ejecución de las pruebas, sino que también busca incorporar mecanismos de mejora continua para las mismas.

4.2.4.1. Etapa de Verificación local

Como se indica en la sección 3.2.2.1, se propone la ejecución de pruebas unitarias las 24 horas del día. A partir de esto, Saff y Ernst proponen la ejecución de pruebas unitarias y de integración en segundo plano mientras el desarrollador escribe el código para dar retroalimentación rápido a los desarrolladores con respecto a los errores que introducían inadvertidamente mientras desarrollaban. Esta etapa de V&V busca adaptar los enfoques mencionados a las pruebas unitarias.

PRÁCTICA 4.1.1: PRUEBAS UNITARIAS EN SEGUNDO PLANO

Las pruebas unitarias se ejecutan luego que el código fue terminado. Cuanto más tiempo pase sin que un error sea detectado, mayor será su costo. Quizás el error se introdujo cuando el desarrollador comenzó a escribir el código y entonces es más difícil solucionarlo.

Esta práctica (Tabla 42) propone ejecutar las pruebas unitarias al mismo tiempo que el desarrollador escribe el código en el entorno de desarrollo y de esta manera detectar el error lo antes posible.

Tabla 42. Práctica: pruebas unitarias en segundo plano.

Pruebas unitarias en segundo plano		
Objetivos	Proporcionar a los desarrolladores retroalimentación rápida sobre errores que han introducido inadvertidamente mientras escriben el código fuente.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Detección tardía de errores con pruebas unitarias.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	La PC del desarrollador soporta tareas programadas que pueden ejecutarse en segundo plano.	Existen pruebas unitarias.
Tareas	1. Creación del script de ejecución de pruebas unitarias en segundo plano.	

Tarea 4.1.1.1: Creación del script de ejecución de pruebas unitarias en segundo plano

Esta tarea tiene como objetivo desarrollar un script del tipo bash o powershell o alguna otra tecnología, que permita invocar constantemente a las pruebas unitarias cuando se lo ejecute. El script se detiene cuando una o más pruebas unitarias fallan o cuando el desarrollador lo desee. En el Algoritmo 14 se muestra un ejemplo de un script de ejecución de pruebas en segundo plano, utilizando bash e invocando a la herramienta de construcción maven.

```
#!/bin/bash
PROFILE="$1"

n=0
while (( $n -ne 0 ))
do
      mvn clean install -P $PROFILE
      if [[ "$?" -ne 0 ]]
            n=$?
done
exit $((n))
```

Algoritmo 14. Script de ejecución de pruebas unitarias en segundo plano.

Como se observa en el Algoritmo 14, mientras el resultado de la ejecución de las pruebas mediante maven sea exitoso, entonces se volverán a ejecutan las mismas, con una compilación previa (comando mvn clean install). En caso de que las pruebas fallen (-ne 0), el script se detendrá mostrando el resultado de las pruebas que fallaron.

El desarrollador puede además configurar una tarea dentro de la computadora que ejecute este script al mismo tiempo que escribe el código en su entorno de desarrollo integrado.

4.2.4.2. Etapa de Construcción

Detener las actividades de desarrollo para esperar retroalimentación del servidor de Integración Continua hace más lento el ritmo de trabajo para todos los integrantes del proyecto [9]. En consecuencia, las construcciones que tardan mucho tiempo en completarse a menudo forman algo desfavorable en las prácticas del desarrollo continuo.

La retroalimentación inmediata del servidor de Integración Continua no solo aplica a esta etapa de V&V en este nivel CTIL, sino a todas las posteriores. En este caso, se proponen prácticas relacionadas con la disminución del tiempo de ejecución de pruebas unitarias, como, por ejemplo, su paralelización.

PRÁCTICA 4.2.1: PRUEBAS UNITARIAS EN PARALELO

Si bien ejecutar pruebas unitarias en serie funciona bien la mayoría de las veces, el tiempo de ejecución de las mismas se puede reducir si se ejecutan en paralelo. Esta práctica se presenta en la Tabla 43.

Tabla 43. Práctica: Pruebas unitarias en paralelo.

Pruebas unitarias en paralelo		
Objetivos	Ejecutar pruebas unitarias en paralelo para reducir el tiempo de ejecución de esta fase.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Detección tardía de errores con pruebas unitarias en el servidor de IC.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	El servidor de IC soporta multihilos.	Existen pruebas unitarias.
Tareas	1. Paralelización de pruebas unitarias.	

Tarea 4.2.1.1: Paralelización de pruebas unitarias

La mayoría de los marcos de trabajo de pruebas actuales soportan la paralelización. Además, permiten elegir el criterio de paralelización: por clases, por métodos, por pruebas, etc.

Al ejecutar pruebas en paralelo, todas las variables y funciones que intervienen con una prueba deben estar relacionadas solo con esa prueba, para evitar errores en tiempo de ejecución. Un patrón útil para adaptar las pruebas unitarias a la paralelización, es el patrón AAA[58] [162]. El patrón sugiere que toda prueba unitaria debe estar dividida en tres secciones, donde cada una de ellas tiene un objetivo: preparar precondiciones, ejecutar acciones y verificar resultados esperados. Si todas las pruebas unitarias cumplen con esta estructura, entonces serán aptas para ser paralelizadas.

[58] *Arrange-Act-Assert*

La implementación de la paralelización varía según el marco de trabajo de pruebas. Por ejemplo, JUnit, permite configurar la paralelización utilizando el plugin surefire de Maven en el mismo pom.xml[59].

Por otro lado, TestNG, utiliza archivos XML de suites, donde no solo se indica la cantidad de hilos a utilizar y el tipo de paralelización (por clases, métodos, pruebas, etc.), sino también qué parte del proyecto se desea paralelizar[60]. En el caso de plataformas .NET, el marco de trabajo NUnit, utiliza una extensión llamada PNUnit[61].

PRÁCTICA 4.2.2: ESCALABILIDAD Y RENDIMIENTO DE LA CONSTRUCCIÓN

La escalabilidad indica qué tan capaz es el servidor de Integración Continua para manejar incrementos en la cantidad de código que debe ser construido, mientras que el rendimiento se refiere a la duración de esta etapa [9]. Ambos atributos forman parte del alcance de esta práctica, que se describe en la Tabla 44.

Tabla 44. Práctica: Escalabilidad y rendimiento de la construcción

Escalabilidad y rendimiento de la construcción		
Objetivos	Obtener y analizar métricas de la etapa de construcción, para realizar mejoras de forma continua.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Si el código base se vuelve grande, el servidor de IC no es capaz de manejar este cambio de magnitud y el rendimiento se degrada.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Existen una herramienta de análisis estático del código fuente.
Tareas	1. Definición y obtención de métricas. 2. Análisis de métricas de la etapa de construcción y ejecución de un plan de acción.	

[59] https://www.baeldung.com/maven-junit-parallel-tests
[60] https://testng.org/doc/documentation-main.html#parallel-running
[61] https://nunit.org/docs/2.5.9/pnunit.html

Tarea 4.2.1.1: Definición y obtención de métricas

El primer paso para mejorar la duración del proceso de construcción del sistema es capturando métricas relacionadas [9]. La Tabla 45 muestra algunas métricas comunes que pueden producir un análisis más cualitativo del proceso de construcción. No es necesario obtener todas estas métricas en todas las ejecuciones, pero es un ejercicio útil para asegurarse de los problemas que existen realmente y no desperdiciar tiempo solucionando problemas que no existen.

Tabla 45. Métricas del proceso de construcción [9].

Métrica	Descripción
Tiempo de compilación	El tiempo que lleva compilar el sistema, comparándolo con anteriores compilaciones.
Número de líneas de código fuente	Indica el tamaño del sistema en términos de lo que necesita ser compilado.
Número y tipos de inspecciones.	Todos los diferentes tipos de métricas del código fuente que se están analizando, para eliminar las redundancias o las que no son útiles.
Tiempo total de ejecución de pruebas unitarias y de despliegue.	El tiempo que lleva ejecutar las pruebas unitarias y las de despliegue.
Razón entre construcciones exitosas y fallidas.	La división entre la cantidad de construcciones fallidas sobre el total de construcciones, para determinar este valor.
Tiempo de ejecución del análisis estático del código fuente.	El tiempo que lleva ejecutar todas las inspecciones de código automáticas.
Tiempo total de despliegue.	El tiempo que lleva desplegar el software al ambiente de prueba luego de la construcción.
Tiempo de regeneración de la base de datos.	El tiempo que lleva volver a generar la base de datos.
Uso de recursos del servidor de Integración Continua durante la etapa de construcción.	Conocer el uso de memoria, la velocidad del disco y el procesador durante esta etapa puede mejorar su rendimiento.

Métrica	Descripción
Carga del sistema de control de versiones.	Ayuda a determinar cuánto tiempo lleva verificar y/o actualizar el repositorio desde el servidor de Integración Continua durante la etapa de construcción y si el ancho de banda de la red, el procesador, la memoria o las unidades de disco son adecuadas.

Tarea 4.2.1.2: Análisis de métricas de la etapa de construcción y ejecución de un plan de acción.

Una vez que las métricas han sido definidas y obtenidas después de ejecutar algunas construcciones, se procede con el análisis de las mismas. Duvall et al. [9], proponen una matriz de mejoras para la duración de la etapa de construcción en el servidor de Integración Continua, la cual puede utilizarse como una guía general para determinar los planes de acciones. Dicha matriz es presentada en la Fig. 54. Las tácticas de mejora presentadas en ella son priorizadas utilizando el siguiente criterio: escalabilidad, rendimiento y dificultad de implementación.

Muchas soluciones dependen del tamaño del código base y ciertos procesos de construcción automáticos que consumen tiempo de ejecución. Es conveniente documentar el enfoque y la justificación para referirse a la próxima vez en que se busque reducir la duración del proceso de construcción.

Táctica de Mejora	Prioridad	Escalabilidad	Rendimiento	Dificultad
Utilizar un servidor de CI dedicado.	1	↑	↑	↓
Incrementar la capacidad del hardware del servidor de CI.	2	↑	—	↓
Mejorar el rendimiento de las pruebas unitarias y despliegue.	3	↓	↑	—
Ordenar las tareas dentro de la etapa de construcción.	4	↓	—	↓
Optimizar la infraestructura.	5	—	—	↑
Optimizar el proceso de construcción.	6	↓	—	↓
Construir componentes del sistema de forma separada.	7	↓	—	—
Mejorar el rendimiento del análisis estático del código.	8	↓	↑	↓
Realizar construcciones distribuidas.	9	↑	↑	↑

Impacto en la escalabilidad, rendimiento y dificultad: ALTO ↑ MEDIO — BAJO ↓

Fig. 54. Matriz de mejoras para la duración del proceso de construcción [9].

Una vez hecho el análisis con las métricas obtenidas y una estrategia para el plan de acción utilizando las tácticas de mejora, se procede con la implementación del plan.

Sin embargo, no es suficiente con mejorar una sola vez el rendimiento del proceso de construcción, es necesario reevaluar, es decir, someter el proceso a mejora continua [3].

4.2.4.3. Etapa de Pruebas funcionales

Un Conductor de Despliegue lento puede ser una de las principales razones por la cual los desarrolladores no envían código al repositorio de control de versiones con frecuencia [9]. En cualquier proyecto de desarrollo, si existen pruebas funcionales como parte del Conductor de Despliegue o del flujo de Integración Continua, las mismas son la causa principal de la lentitud [7], [9], [11], [12], [14]. Sin embargo, no se puede disminuir la cobertura de pruebas funcionales, ya que, al hacerlo, se disminuye la confiabilidad.

De este modo, tanto la literatura académica [11] como la industria [15] han generado propuestas para hacer frente a los problemas de pruebas que llevan mucho tiempo de ejecución. Entre estas propuestas se encuentran la paralelización de pruebas funcionales, uso de API, comparación de imágenes, gestión de datos de pruebas y rotación de navegadores (esta última

se incorpora luego del quinto ciclo de la validación descrito en la sección 6.2.6). En este nivel CTIL, dichas propuestas son presentadas como prácticas para esta etapa de V&V.

PRÁCTICA 4.3.1: PRUEBAS FUNCIONALES EN PARALELO

La ejecución de pruebas funcionales automatizadas en paralelo (en lugar de en serie), disminuye el tiempo de ejecución de las mismas, ya que se distribuye a través de diferentes computadoras o procesadores en la misma [11]. Si se cuenta con mucho hardware, o con un servicio que provee gran cantidad de dispositivos y navegadores en la nube, esta es la mejor solución para disminuirlos tiempos de ejecución del Conductor de Despliegue [176]. Sin embargo, es la más costosa [177]. El objetivo de esta práctica, los problemas que resuelve y los requisitos, se muestran en la Tabla 46, junto con las tareas necesarias para su implementación.

Tabla 46. Práctica: Pruebas funcionales en paralelo.

Pruebas funcionales en paralelo		
Objetivos	Ejecutar pruebas funcionales en paralelo para reducir el tiempo de ejecución de esta fase.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas funcionales que tardan mucho tiempo de ejecución.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Mucho hardware, muchos dispositivos físicos, o servicios en la nube que lo proveen.	Existen pruebas funcionales con un marco de trabajo que soporta paralelización.
Tareas	1. Paralelización de pruebas funcionales.	

Tarea 4.3.1.1: Paralelización de pruebas funcionales

Como se menciona en la práctica de paralelización de pruebas unitarias, la mayoría de los marcos de trabajo de pruebas actuales soportan la paralelización por clases, métodos, pruebas, grupos y lotes de pruebas. Asimismo, como se detalló en la sección 4.2.4.2, cada marco de trabajo de prueba tiene su propia configuración para permitir paralelización. Sin embargo, a diferencia de las pruebas unitarias, las pruebas funcionales utilizan mucho hardware cuando no se ejecutan en serie.

Existen varias alternativas, con diferentes prestaciones según su costo, que son utilizadas para construir una infraestructura de paralelización de pruebas funcionales:

- Clúster o nube privada: es un conjunto de computadoras o servidores conectados a la misma red privada, que se comportan como si fuesen un único servidor. Cuanto mayor sea la cantidad de máquinas conectadas a la red, mayor serán las prestaciones de la infraestructura de paralelización. Además del costo inicial para adquirir los equipos, requiere costos de mantenimiento de la infraestructura.
- Virtualización: Consiste en utilizar una o varias computadoras conectadas entre sí, con altas capacidades de memoria RAM y procesador, para ejecutar máquinas virtuales utilizando software especializado como VMware[62] o VirtualBox[63] y ejecutar las pruebas en las mismas. El costo varía en base a los recursos de la/s computadora/s utilizadas.
- Uso de contenedores: Un contenedor es similar a una máquina virtual, con la diferencia de que, en lugar de albergar un sistema operativo completo, lo que hace es compartir los recursos del propio sistema operativo "host" sobre el que se ejecuta. Son más livianos que las máquinas virtuales y se generan solo cuando van a ser utilizados, es decir, de forma dinámica. De forma similar a la virtualización, se requiere de una o varias computadoras conectadas entre sí con mucho hardware. Como utiliza menos recursos, es más barato que utilizar virtualización, pero requiere mayor configuración y se limita a sistemas operativos Windows y Linux. El software más utilizado es Docker[64].
- Servicios en la nube o cloud público: Consiste en plataformas web que brindan infraestructura que soporta diferentes dispositivos móviles, navegadores y sistemas operativos, para que puedan utilizarse cuando se lo desee. Su costo varía en relación al uso y el tamaño de la infraestructura que se adquiere y no se paga por mantenimiento. Entre las opciones más comunes se encuentran: BrowserStack[65], CrossBrowserTesting[66], Sauce Labs[67] y Experitest[68]. Esta opción fue incorporada como sugerencia en la quinta validación del modelo (sección 6.2.6.3).

[62] https://www.vmware.com/
[63] https://www.virtualbox.org/
[64] https://www.docker.com/
[65] https://www.browserstack.com/
[66] https://crossbrowsertesting.com/
[67] https://saucelabs.com/
[68] https://experitest.com/

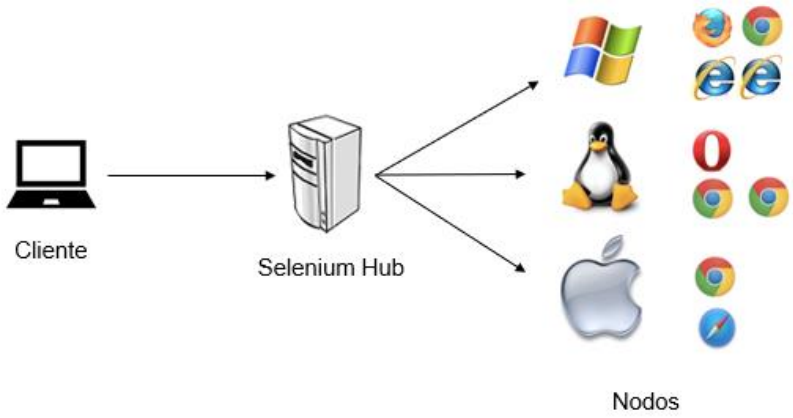


Fig. 55. Arquitectura de Selenium Grid.

La utilización de cualquiera de las infraestructuras mencionadas para ejecutar pruebas funcionales en paralelo se denomina **Grid**. Algunas herramientas como Selenium Grid[69], facilitan el despliegue de esta infraestructura utilizando la arquitectura de concentrador-nodos. El concentrador (Selenium Hub) es el servidor desde el que se ejecutan las pruebas automatizadas y los nodos (Selenium Nodes) son cada uno de los dispositivos (físicos, virtualizados, en contenedores o prestados por un servicio en la nube) donde se ejecutan las mismas. Esta arquitectura se muestra en la Fig. 55. Todo esto se genera a partir de las instrucciones ejecutadas desde un cliente (máquina local o servidor de Integración Continua).

PRÁCTICA 4.3.2: SELECCIÓN DE PRUEBAS A EJECUTAR

En la mayoría de los flujos de Integración Continua, la etapa de pruebas funcionales contiene un conjunto de pruebas críticas seguidas por pruebas secundarias que llevan más tiempo de ejecución. Si en lugar de ejecutar estas pruebas secundarias, se ejecutan solo las pruebas que verifican el código que ha sido modificado, entonces se reduce el tiempo de ejecución considerablemente [178]. Este proceso se denomina selección de pruebas a ejecutar y la Tabla 47 muestra la práctica que lo contiene.

Tabla 47. Práctica: Selección de pruebas a ejecutar.

Selección de pruebas a ejecutar	
Objetivos	Ejecutar solamente las pruebas que verifican el código que ha sido introducido.

[69] https://www.selenium.dev/documentation/en/grid/

Selección de pruebas a ejecutar		
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas funcionales que tardan mucho tiempo de ejecución.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	El servidor de Integración Continua soporta herramientas o plug-ins que permiten leer los mensajes en las confirmaciones de cambios.	Existen pruebas funcionales agrupadas según funcionalidades del sistema.
	El marco de trabajo de pruebas soporta la construcción de lotes de pruebas de forma dinámica.	
Tareas	1. Lectura de mensajes de los cambios que son subidos al repositorio de control de versiones. 2. Ejecución de pruebas en base a los mensajes de los cambios que son subidos al repositorio de control de versiones.	

Tarea 4.3.2.1: Lectura de mensajes de los cambios que son subidos al repositorio de control de versiones

Para ejecutar un conjunto de pruebas específicas, es necesario que estén agrupadas por funcionalidad, siguiendo la práctica de segmentación de pruebas funcionales mencionada en 4.2.2.3. Además, se necesita un mecanismo que permita identificar a qué funcionalidad del sistema afectan los cambios introducidos, para así ejecutar las pruebas correctas. Una de las formas de hacerlo, es mediante la lectura de los mensajes que se incluyen cuando los desarrolladores suben los cambios al repositorio de control de versiones. Para ello, se debe incorporar a los procesos de desarrollo que los desarrolladores agreguen una etiqueta en la confirmación de cambios, que permita realizar una trazabilidad entre la misma y la funcionalidad del sistema afectada (ej: “Requisito 12345 - Gestión de Stock – Eliminación del campo del servicio web updateStock).

La mayoría de los sistemas de control de versiones, soportan comandos para tomar el mensaje del último conjunto de cambios subidos al repositorio. Por ejemplo, el Algoritmo 15 muestra el comando utilizado para este propósito con el sistema de control de versiones git.

```
git log -p -1
```

Algoritmo 15. Comando git para obtener el mensaje de los últimos cambios subidos al repositorio.

Si se ejecutase este comando en el servidor de Integración Continua antes de las pruebas funcionales, entonces se podría obtener el mensaje de los últimos cambios subidos al repositorio, con la etiqueta que indica la parte o funcionalidad del sistema que ha sido modificada.

En algunos servidores de Integración Continua, se deben instalar herramientas para acceder a esta funcionalidad. Por ejemplo, en Jenkins + Git, es necesario instalar un plugin como el Environment Injector[70]. El último paso, consiste en inyectar la etiqueta obtenida del mensaje, en la ejecución de pruebas.

Tarea 4.3.2.2: Ejecución de pruebas en base a los mensajes de los cambios subidos al repositorio de control de versiones.

Una vez que se cuente con una variable que indique el área del sistema que fue afectada por los cambios introducidos, se puede construir un lote dinámicamente con las pruebas relacionadas. La variable puede ser tomada a través de la herramienta que se utiliza para construir el proyecto, como Maven, Ant, NAnt, MSBuild y Rake (descritas en la sección 2.3.1)

La mayoría de los marcos de trabajo de pruebas soportan la construcción de lotes de pruebas dinámicos. En el Algoritmo 16 se muestra un ejemplo en Java para el marco de trabajo TestNG[71].

Si el marco de trabajo de pruebas no permite la creación de lotes dinámicos, existe una alternativa que consiste en crear lotes por cada grupo de pruebas. Luego, en base al valor de la variable tomada del mensaje de los cambios subidos al repositorio, se ejecuta el lote correspondiente.

[70] https://plugins.jenkins.io/envinject/

[71] https://testng.org/doc/documentation-main.html#running-testng-programmatically

```
public static void generateSuite() {

  //Se obtiene el grupo de pruebas a ejecutar
  String group = System.getProperty("group");

  //Se construye el lote
  XmlSuite suite = new XmlSuite();
  suite.setName("Custom Suite");
  XmlTest test = new XmlTest(suite);
  test.setName(group + " Tests");
  XmlRun xmlRun = new XmlRun();
  xmlRun.onInclude(test.getName());

  List<XmlPackage> packages = new ArrayList<XmlPackage>();
  packages.add(new XmlPackage("com.product.test." + group));
  test.setXmlPackages(packages);
  List<XmlSuite> suites = new ArrayList<XmlSuite>();
  suites.add(suite);
  TestNG tng = new TestNG();
  tng.setXmlSuites(suites);
  TestListener tla = new TestListener();
  tng.addListener((ITestNGListener) tla);
  tng.run();

}
```

Algoritmo 16. Creación de un lote de pruebas dinámico en TestNG.

Finalmente, si no fue posible detectar una etiqueta que tenga una relación con un grupo de pruebas específicos, es importante continuar ejecutando el lote de pruebas regular, que forma parte de las pruebas secundarias, como se muestra en la Fig. 56.

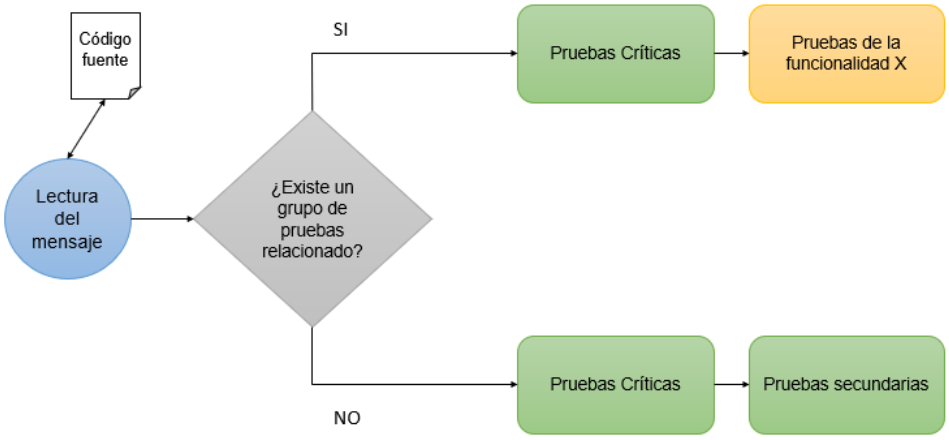


Fig. 56. Flujo de ejecución de pruebas específicas en base a mensajes de los cambios subidos al repositorio.

PRÁCTICA 4.3.3: USO DE API PARA LA EJECUCIÓN DE PRECONDICIONES

En la mayoría de los casos, las pruebas de GUI tienen precondiciones que requieren la ejecución de pasos de inicialización o configuración mediante la misma GUI de la aplicación. La utilización de las API para ejecutar las precondiciones reduce la duración de las pruebas ya que son considerablemente más rápidas en comparación con realizarlo a través de la GUI [11], [179]. Esta práctica de utilizar API para la ejecución de precondiciones se presenta en la Tabla 48.

Tabla 48. Práctica: Uso de API para la ejecución de precondiciones.

Uso de API para la ejecución de precondiciones		
Objetivos	Disminuir el tiempo de ejecución de pruebas de GUI utilizando llamadas a API.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas funcionales de GUI que tardan mucho tiempo de ejecución.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Se tiene acceso a las API que son utilizadas por la GUI.	No aplica.
Tareas	1. Reemplazo de precondiciones en la GUI por llamadas a las API.	

Tarea 4.3.3.1: Reemplazo de precondiciones en la GUI por llamadas a las API.

Un caso de prueba está formado por precondiciones, un conjunto de pasos, datos de prueba, postcondiciones y verificación de resultados esperados [82]. Sin embargo, de esos componentes, las precondiciones es uno de los que no forman parte del objetivo real del caso de prueba, sino que representan un estado en el que debe estar el sistema previamente para realizar los pasos y las verificaciones que si son pertinentes a la prueba. Por ejemplo, en un sistema de comercio electrónico, si se tiene un caso de prueba cuyo objetivo es verificar que un usuario tenga guardadas las compras hechas en el pasado, se tienen dos precondiciones: que el usuario haya iniciado sesión y que el mismo haya realizado mínimamente una compra. En una prueba de GUI convencional, se utiliza un driver como Selenium para simular un usuario ingresando al sitio a través del navegador web. Luego, el mismo puede ya tener hechas algunas compras efectuadas como parte de otro caso de prueba, o puede tener que realizarlas como parte de la misma prueba, para continuar luego con las verificaciones correspondientes. En este ejemplo, se

observa que el tiempo de ejecución de una prueba aumenta considerablemente debido a las precondiciones que se tienen que ejecutar previamente.

De acuerdo a Sinisalo [179], si estas precondiciones se ejecutaran utilizando una API, el tiempo de la prueba se reduciría considerablemente. De este modo, si se tiene acceso a las API que utiliza el sistema para realizar las acciones, entonces todas las precondiciones de las pruebas de GUI pueden reemplazarse con llamadas a estas API. El ejemplo presentado se muestra en el Algoritmo 17.

```
LoginService loginService = new LoginService();
PurchaseService purchaseService = new PurchaseService();


@Test
public void verifyPastShoping (String username, Strin password) {
      HTTPResponse logingResponse = loginService.login(username,
password);
      Cookie loginCookie = new Cookie("token", logingResponse.getToken());
      driver.manage().addCookie(loginCookie);
      HTTPResponse purchaseResponse =
      purchaseService.login(username, password);
      if (purchaseResponse.statusCode() != 200)
            throw new SkipException("Preconditions were not met");
      driver.findElement(By.id("old_shopping")).click();
      List<WebElement> previousPurchases =
            driver.findElements(By.id("past_purchase"));
      Assert.assertTrue(previousPurchases.size > 0,
            "Past purchases are not being shown");
}
```

Algoritmo 17. Ejemplo del uso de API para ejecutar precondiciones en pruebas de GUI.

PRÁCTICA 4.3.4: ROTACIÓN DE NAVEGADORES

En lugar de ejecutar las pruebas repetitivamente por cada navegador, si se ejecutan en paralelo utilizando un navegador diferente por cada prueba, entonces, se disminuirá el tiempo de ejecución de las mismas considerablemente, según la cantidad de navegadores que se utilicen [179]. Esta práctica se presenta en la Tabla 49.

Tabla 49. Práctica: Rotación de navegadores.

Rotación de navegadores	
Objetivos	Disminuir el tiempo de ejecución de pruebas de GUI utilizando varios navegadores en una única ejecución.

Rotación de navegadores		
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas funcionales de GUI que tardan mucho tiempo de ejecución.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Algunas o todas de las pruebas funcionales de GUI se ejecutan en una plataforma web.	Las pruebas funcionales de GUI se ejecutan en paralelo.
	Es requisito probar el software en más de un navegador.	Se posee una infraestructura compuesta por varios servidores físicos o virtuales con diferentes navegadores.
Tareas	1. Configuración de la ejecución para distribuir los hilos entre diferentes navegadores.	

Tarea 4.3.4.1: Configuración de la ejecución para distribuir los hilos entre diferentes navegadores

Rotar los navegadores entre las pruebas en paralelo logra gradualmente la misma cobertura que ejecutar las mismas pruebas en un solo navegador. Por ejemplo, si la ejecución de un conjunto de pruebas funcionales web tarda 40 minutos, porque se ejecutan pruebas que tardan 10 minutos en 4 navegadores, entonces, las pruebas se ejecutarían en diferentes navegadores una única vez y el tiempo se reducirá a 10 minutos. Otra alternativa es usar una solución intermedia, donde en lugar de ejecutar en todos los navegadores de una sola vez, pueden usarse la mitad de ellos en una ejecución y la otra mitad en una segunda. De esta manera, el tiempo de ejecución se reduciría a la mitad.

En el Algoritmo 18 se muestra un ejemplo de una clase java que gestiona los nodos de un grid de Selenium, que permite la distribución de las pruebas en paralelo a través de los mismos.

```
public class NodeHandler {

      private static NodeHandler instance;
      private static synchronized List<String> availableNodes;
      private static synchronized List<String> busyNodes;
      private static synchronized Map<Long, String> nodes;

      private static String nodeOne = "http://10.0.0.22:4546/wd/hub";
      private static String nodeTwo = "http://10.0.0.22:4547/wd/hub";
      private static String nodeThree = "http://10.0.0.22:4548/wd/hub";
      private static String nodeFour = "http://10.0.0.22:4549/wd/hub";

      private NodeHandler() {
            availableNodes = new ArrayList<String>();
            nodes = new HashMap<Long, String>();
            availableNodes.add(nodeOne);
            availableNodes.add(nodeTwo);
            availableNodes.add(nodeThree);
            availableNodes.add(nodeFour);
      }

      public static synchronized String useNode(Long threadId) {
            String nodeUrl = availableNodes.get(0);
            nodes.put(threadId, nodeUrl);
            busyNodes.add(nodeUrl);
            availableNodes.remove(0);
            return nodeUrl;
      }

      public static synchronized void releaseNode(Long threadId) {
            String nodeUrl = nodes.get(threadId);
            availableNodes.add(nodeUrl);
            busyNodes.remove(nodeUrl);
            nodes.values().remove(threadId);
      }

      static NodeHandler getInstance() {
    if (instance == null) {
      synchronized (NodeHandler.class) {
        if (instance == null) instance = new NodeHandler();
      }
    }
    return instance;
  }
}
```

Algoritmo 18. Gestión de nodos para ejecutar pruebas utilizando rotación de navegadores.

PRÁCTICA 4.3.5: COMPARACIÓN DE IMÁGENES

La comparación de imágenes es una técnica muy utilizada para la detección de incompatibilidades web entre diferentes navegadores [123]. Pero debido a la naturaleza de cada

navegador, siempre se encontraban diferencias en las comparaciones [120]. Sin embargo, si se utiliza para realizar una comparación de una misma página o ventana, en un mismo navegador, pero entre diferentes ambientes, entonces los resultados son precisos. De esta manera, las diferencias detectadas por el algoritmo que compara las imágenes son utilizadas para detectar defectos antes de llegar a producción. Esta práctica es presentada en la Tabla 50.

Tabla 50. Práctica: Comparación de imágenes.

Comparación de imágenes		
Objetivos	Detectar errores introducidos en la GUI de una aplicación, comparando una captura de pantalla de la misma en un ambiente de prueba contra otra captura de pantalla de producción.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Pruebas funcionales manuales de GUI que tardan mucho tiempo de ejecución.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	El software se despliega en uno o más ambientes de pruebas antes de producción.	Se ejecutan regresiones manuales de GUI web, de escritorio o en dispositivos móviles.
Tareas	1. Implementar un algoritmo de comparación de imágenes. 2. Escribir los casos de pruebas automatizados que ejecuten el algoritmo de comparación de imágenes.	

Tarea 4.3.5.1: Implementar un algoritmo de comparación de imágenes

La técnica consiste en complementar el proceso de pruebas funcionales sobre un sitio web, con un algoritmo de comparación de imágenes automatizado y de esta manera acelerar la detección de diferencias entre diferentes versiones de la aplicación.

Existen diferentes algoritmos de comparación de imágenes, pero el más utilizado es el análisis visual mediante comparación de pixeles [124].

La entrada del algoritmo es un par de imágenes que son las capturas de pantallas de una página o vista de la aplicación. Una de las imágenes es la base, que representa la versión estable en producción, mientras que la otra es la imagen con la nueva versión de la aplicación que está en ejecución en un ambiente de pruebas y contiene los cambios recientes del código fuente.

```
private Result compareImages(Image image1, Image image2, String name) {

  BufferedImage img1 = image1.getImg();
  BufferedImage img2 = image2.getImg();
  boolean result = true;
  int cont = 0;
  int pxDiff = 0;
  BufferedImage heatMapResult = null;
  if (img1.getWidth() == img2.getWidth()
     && img1.getHeight() == img2.getHeight()) {
    heatMapResult = new BufferedImage(img1.getWidth(),
      img1.getHeight(), BufferedImage.TYPE_INT_RGB);
    for (int x = 0; x < img1.getWidth(); x++) {
      for (int y = 0; y < img1.getHeight(); y++) {
        if (img1.getRGB(x, y) != img2.getRGB(x, y)) {
          result = false;
          heatMapResult = createHeatMap(heatMapResult,x,y);
          pxDiff++;
        }
        cont++;
      }
    }
  } else {
    result = false;
  }
  String imgPath = reportFilePath + fileSeparator + name + ".png";
  ImageIO.write(heatMapResult, "png", new File(imgPath));
  int pxEquals = cont - pxDiff;
  return new Result(name, image1.getName(), image2.getName(), pxEquals,
      pxDiff, result);
}
```

Algoritmo 19. Algoritmo de comparación de imágenes.

El algoritmo toma el par de imágenes y ejecuta los siguientes pasos (ver Algoritmo 19):

1. Se verifica que ambas imágenes tengan las mismas dimensiones. Si no son iguales, se ignora la comparación y se obtiene un fallo y termina el proceso. Si son iguales, se continua con el próximo paso.
2. Se obtiene una matriz de pixeles de cada imagen.
3. Se recorren todos los pixeles de la imagen base uno por uno, comparándolos con su recíproco en la otra imagen. En caso de ser diferentes, se guarda la posición de los pixeles diferentes en una nueva matriz resultado.
4. Si no hubo diferencias, la prueba de compatibilidad ha sido completada con éxito. En caso de fallo, se genera una tercera imagen resultado de la comparación utilizando la matriz resultado. La imagen es pintada con puntos en cada posición correspondiente a los pixeles diferentes. Esta imagen es guardada con un formato de mapa de calor junto al par de capturas que fueron analizadas.

Tarea 4.3.5.2: Escribir los casos de pruebas automatizados que ejecuten el algoritmo de comparación de imágenes

Una vez que el algoritmo de comparación de imágenes fue implementado, el siguiente paso consiste en escribir los scripts de pruebas que naveguen hasta una cierta sección del sitio web o ventana de la aplicación, en cada ambiente y tome las capturas de pantallas correspondientes para luego ejecutar el algoritmo.

Para realizar esos pasos, se puede utilizar la misma herramienta con las que se construyen las pruebas funcionales automatizadas. Por ejemplo, según el tipo de plataforma, para web, se puede utilizar Selenium WebDriver[72], para escritorio, alguna herramienta como Marathon[73] y para dispositivos móviles, Appium[74].

```
private String pageOne = prod.environment.com
private String pageTwo = staging.environment.com
private ImageComparator comparator = new ImageComparator();

@Test
public void compareHomePage() {
  driver.manage().window().fullscreen();
  driver.get(pageOne);
  Image pageOneScreenshot = comparator.capture("homePageProd");

  driver.get(pageTwo);
  Image pageTwoScreenshot = comparator.capture("homePageStaging");

  Result result = comparator.compare(pageOneScreenshot,
    pageTwoScreenshot, "homeResult");

  assertTrue(result.isResult(),"The comparison failed between " +
    + "staging and prod in the homepage");
}
```

Algoritmo 20. Ejemplo de navegación, toma de capturas de pantallas y comparación de imágenes.

Una vez tomadas las capturas de pantallas, se deben guardar en un directorio para luego utilizarlas en el reporte final de comparación, junto con el mapa de calor resultante que contiene las diferencias.

[72] https://www.selenium.dev/documentation/en/webdriver/
[73] https://marathontesting.com/
[74] http://appium.io/

En el Algoritmo 20 se muestra un ejemplo de la navegación hacia una determinada página, en dos ambientes diferentes (producción y staging), tomando capturas de pantalla y ejecutando la comparación de imágenes.

Finalmente, en la se muestra un ejemplo de un resultado de la comparación de imágenes utilizando los algoritmos mencionados anteriormente.

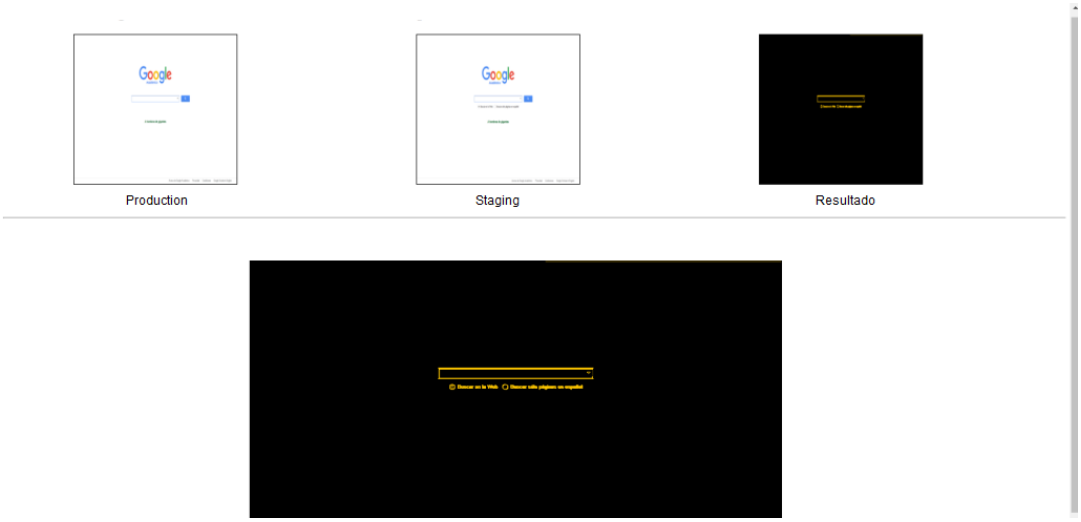


Fig. 57. Ejemplo de un resultado de la comparación de imágenes.

PRÁCTICA 4.3.6: MONITORIZACIÓN CONTINUA DE PRUEBAS FUNCIONALES

En los entornos de desarrollo continuo, existe la oportunidad de observar varios parámetros de calidad en todo momento [18], [180]. De esta manera, al igual que en la etapa de construcción, es posible medir la escalabilidad y el rendimiento de la etapa de pruebas funcionales [9]. El objetivo de esta práctica es medir dichos atributos en la etapa de pruebas funcionales y la misma se presenta en la Tabla 51.

Tabla 51. Práctica: Monitorización continua de pruebas funcionales.

Monitorización continua de pruebas funcionales		
Objetivos	Obtener y analizar métricas durante la ejecución de las pruebas funcionales, para realizar mejoras de forma continua.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Gran cantidad de pruebas funcionales.	Tiempos de ejecución elevados.
	Pruebas funcionales no optimizadas.	

Monitorización continua de pruebas funcionales		
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	No aplica.
Tareas	1. Definición y obtención de métricas. 2. Análisis de métricas de la etapa de pruebas funcionales y ejecución de un plan de acción.	

Tarea 4.3.6.1: Definición y obtención de métricas

Al igual que en la a de construcción el primer paso para mejorar la duración y el rendimiento de la ejecución de pruebas funcionales es la captura de métricas [9] como las de la Tabla 52.

Tabla 52. Métricas para monitorización continua de pruebas funcionales.

Métrica	Descripción
Tiempo de compilación.	El tiempo que lleva compilar el código de las pruebas funcionales, comparándolo con anteriores compilaciones.
Tiempo total de ejecución.	El tiempo total de ejecución de todas las pruebas funcionales.
Tiempo de ejecución de las pruebas por categoría o grupo.	El tiempo total de ejecución de las pruebas funcionales por cada grupo (pruebas de humo, función X, función Y, regresiones, etc.)
Razón entre ejecuciones exitosas y fallidas.	La división entre la cantidad de ejecuciones de pruebas funcionales fallidas sobre el total de ejecuciones, para determinar este valor.
Uso de recursos del servidor de Integración Continua durante la ejecución de pruebas funcionales.	Conocer el uso de memoria, la velocidad del disco y el procesador durante la ejecución de pruebas funcionales puede mejorar su rendimiento.
Cantidad de pruebas no deterministas por ejecución (añadida luego de la etapa de validación del modelo en el apartado 6.2.6.3)	Medir la cantidad de pruebas no deterministas por cada ejecución de pruebas funcionales.

Para la obtención de métricas se pueden utilizar tanto plugins del servidor de Integración Continua que se esté utilizando, como scripts que se construyen utilizando el mismo lenguaje que se usa para desarrollar las pruebas funcionales.

Tarea 4.3.6.2: Análisis de métricas de la etapa de pruebas funcionales y ejecución de un plan de acción.

Una vez que las métricas han obtenidas mediante el sistema de monitorización continua, se procede con el análisis de las mismas. Al igual que en la etapa de construcción, para diseñar los planes de acciones, se puede utilizar la matriz de Duvall et al. [9], que se presenta en la Fig. 54. Una vez hecho el análisis con las métricas obtenidas y una estrategia para el plan de acción utilizando las tácticas de mejora, se procede con la implementación del plan.

```
private static final long DURATION_TIME_THRESHOLD = 2400000;
private static final double FAILURE_RATIO_THRESHOLD = 0.5;
private static final int FLAKY_TESTS_THRESHOLD = 10;


public List<MonitoringError> monitoringSystem() {
      List<MonitoringError> errors = new ArrayList<MonitoringError>();

      long durationTime = getTotalDurationTime();
      if (durationTime > DURATION_TIME_THRESHOLD)
            errors.add(new MonitoringError("Duration",durationTime));

      double ratio = getFailureRatio();
      if (ratio > FAILURE_RATIO_THRESHOLD)
            errors.add(new MonitoringError("Failure Ratio", ratio));

      int previousFlakyTests = getPreviousFlakyTestsCount();
      int totalFlakyTests = previousFlakyTests + FLAKY_TESTS_THRESHOLD;
      int newFlakyTests = getCurrentFlakyTestsCount();
      if (newFlakyTests > totalFlakyTests)
            errors.add(new MonitoringError("Flaky Tests", newFlakyTests));

      return errors;
}
```

Algoritmo 21. Ejemplo de monitorización en pruebas funcionales.

También se pueden incorporar umbrales para cada métrica definidas en la tarea anterior, de tal modo que el sistema de monitorización dispare algún tipo de notificación cuando se encuentra un incidente. Un ejemplo se muestra en el Algoritmo 21.

4.2.4.4. Etapa de Pruebas no funcionales

Las pruebas no funcionales como las pruebas de rendimiento tienen limitaciones cuando se trata de ejecutarlo dentro de entornos de desarrollo continuo. Para que una prueba no funcional sea confiable, la infraestructura en la que se ejecutan las pruebas debe ser consistentemente apropiada [9]. Algunas requieren que el entorno de tiempo de ejecución esté especialmente provisto para soportar el propósito de las pruebas. Sin embargo, el entorno no es la única limitación cuando se trata de ejecutarlas en un Conductor de Despliegue. El tiempo de ejecución de estas pruebas también es un impedimento, además del costo involucrado.

A pesar de los desafíos, existen prácticas que hacen que para incluir pruebas no funcionales confiables y automatizadas en los Conductores de Despliegue. Hasta el momento, dentro de esta etapa en los CTIL anteriores se abarcaban pruebas de capacidad que requerían intervención manual. En este CTIL, el objetivo es automatizar las pruebas de capacidad para que sean incorporados en el Conductores de Despliegue como una fase totalmente automatizada.

PRÁCTICA 4.4.1: AUTOMATIZACIÓN DE PRUEBAS DE CAPACIDAD

Esta práctica (Tabla 53) consiste en escribir scripts de pruebas de capacidad que puedan incluirse como una etapa automatizada dentro del Conductores de Despliegue [7]. Es decir, incluir la ejecución de un lote de pruebas de capacidad cuando se introducen cambios, luego de la ejecución de pruebas unitarias, funcionales, etc.

Tabla 53. Práctica: Automatización de pruebas de capacidad.

Automatización de pruebas de capacidad		
Objetivos	Detectar reducciones en la capacidad del sistema tan pronto como sea posible, es decir, cuando los cambios fueron en el código fueron recientemente introducidos y así solucionarlos inmediatamente.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Las pruebas de capacidad tienen intervención manual.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	No aplica.	Contar con un ambiente de pruebas de performance.
Tareas	1. Implementación de pruebas de capacidad automatizadas	

Tarea 4.4.1.1: Implementación de pruebas de capacidad automatizadas

Las pruebas de capacidad deberían [7]:

- Verificar escenarios del mundo real.
- Tener un umbral predefinido para el “éxito”, para determinar el resultado de la ejecución de una prueba.
- Ser de corta duración.
- Ser robustos con respecto a los cambios, para evitar retrabajo constante y mantenimiento.
- Ser capaces de componer o formar escenarios más complejos y de gran escala.
- Ser repetibles y capaces de ser ejecutados de forma secuencial y en paralelo.

Lograr todos estos objetivos sin retrasar el progreso del desarrollo de la aplicación principal no es una tarea sencilla. Una estrategia es tomar unas de las pruebas de funcionales existentes y adaptarlas para convertirlas en pruebas de capacidad [181].

Sin embargo, uno de los aspectos más importantes a considerar es si las interacciones con el sistema se realizan mediante la GUI o mediante una servicio o API. Si se está desarrollando un sistema que es o será usado por miles de usuarios, será imposible generar carga al sistema mediante interacciones con la GUI, ya que, para simular un escenario real, se necesitaran miles de máquinas clientes interactuando con el sistema. En cambio, utilizando los servicios web o las API, esto no es un problema. La Fig. 58 muestra los puntos potenciales para pruebas de capacidad. Esta recomendación surge luego de la validación del modelo en el quinto ciclo (sección 6.2.6.3).

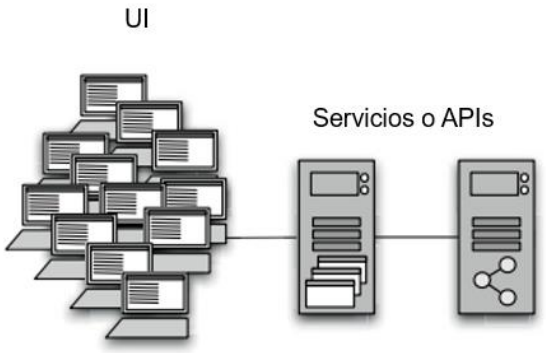


Fig. 58. Puntos potenciales de inyección para pruebas de capacidad.

Tomando una prueba de API que represente un escenario real, se la puede seleccionar para un lote de pruebas de capacidad utilizando un grupo especial, como se muestra en el Algoritmo 22. El paso final es crear un nuevo lote de pruebas, utilizando el grupo agregado y un número de hilos que represente el umbral aceptado, es decir, con el que se espera que la prueba de capacidad no falle (por ejemplo: 500 hilos, representando 500 llamadas a la API).

```
@Test(groups = {"smoke", "capacity"}
public void verifyPurchase() {
  LoginResponse loginResponse = loginService.login(USERNAME, PASSWORD);
  Assert.assertEquals(loginResponse.getStatusCode(), 200,
    "The login was not done correctly");
  Assert.assertTrue(StringUtils.isNotEmpty(loginResponse.getToken()),
    "The login token is empty");

  PurchaseService purchaseService = new PurchaseService(
    loginResponse.getToken());
  PurchaseResponse purchaseResponse = purchaseService
    .performPurchase(ITEMS);
  Assert.assertEquals(purchaseResponse.getStatusCode(), 200,
    "The purchase was not done correctly");
  Assert.assertEquals(purchaseResponse.getShoppedItems().size(),
    ITEMS.size(), "The items were not purchased correctly");
}
```

Algoritmo 22. Prueba de API utilizada como prueba de capacidad.

4.2.4.5. Etapa de pruebas de aceptación de usuario

En esta última etapa y al igual que en las anteriores, el objetivo del CTIL es reducir los tiempos de ejecución, en particular de las pruebas manuales. En este punto, solo quedan las pruebas exploratorias con intervención humana, ya que el resto fue automatizado.

A continuación se describe una práctica que propone la ejecución de pruebas manuales por usuarios que no forman parte del equipo de desarrollo, con una previa capacitación para hacerlo correctamente, basándose en el principio de colaboración abierta distribuida, también conocido como externalización abierta de tareas[75] [182].

[75] *Crowdsourcing*

PRÁCTICA 4.5.1: PRUEBAS MEDIANTE EXTERNALIZACIÓN ABIERTA DE TAREAS

La ejecución de la etapa de pruebas manuales mediante la técnica de externalización abierta de tareas es posible al permitir que los usuarios evaluadores accedan al sistema utilizando algún servidor público accesible mediante internet, a través de sus navegadores web [11], [183]. Es necesario describir los pasos para ejecutar las pruebas, incluyendo los medios a utilizar, como, por ejemplo: micrófono, notas, etc. Esta práctica se presenta en la Tabla 54.

Tabla 54. Pruebas mediante externalización abierta de tareas.

Objetivos	Someter el sistema en desarrollo a un conjunto de usuarios a través de internet, distribuidos en diferentes sitios geográficamente, para que los mismos puedan ejecutar las pruebas correspondientes, a cambio de algún tipo de beneficio.	
Problemas que resuelve	**Problema individual**	**Problema en Pruebas Continuas**
	Etapa de pruebas manuales de larga duración.	Tiempos de ejecución elevados.
Requisitos	**Requisitos generales**	**Requisitos de Pruebas Continuas**
	Acceso público a través de internet a algún ambiente de prueba.	No aplica.
Tareas	1. Implementación de pruebas mediante externalización abierta de tareas.	

Tarea 4.5.1.1: Implementación de pruebas mediante crowdsourcing.

El flujo de pruebas mediante la colaboración abierta distribuida se muestra en la Fig. 59: la empresa de desarrollo de software hace uso de la plataforma de externalización abierta de tareas para cargar los escenarios de pruebas a ejecutar, como si fueran un conjunto de tareas. Cada tarea consta de una serie de pasos que el usuario debe ejecutar. Es importante que el ambiente de prueba a utilizar por los usuarios esté disponible a través de internet y el mismo debe ser incluido en la lista de pasos. Mediante algún tipo de aviso, publicidad, o promoción, la empresa presenta a los usuarios la disponibilidad de esas tareas a realizar, informando acerca de algún beneficio a recibir al completarlas, sea económico o no.

Cuando los usuarios se registran en la plataforma, tienen a su disponibilidad la lista de tareas de la empresa para realizar. Cuando un usuario selecciona la tarea, la plataforma le muestra los diferentes pasos a ejecutar, que forman parte del escenario de prueba. Asimismo, en la mayoría de los casos, al ser la primera tarea, se presenta un breve instructivo que indica a

los usuarios como usar la plataforma correctamente, mencionando el uso de un micrófono para grabar lo que el usuario expresa durante la ejecución de la prueba.

Fig. 59. Ejecución de pruebas manuales mediante externalización abierta de tareas.

Una vez que el usuario termina con la ejecución de la prueba, tiene la opción de marcar la misma como “exitosa”, “fallida”, o “inconclusa”. Luego, la empresa recibe retroalimentación de la realización de estas tareas mediante la plataforma.

Existen diferentes herramientas que permiten realizar externalización abierta de tareas. Algunas herramientas son de propósito general, como por ejemplo, Amazon Mechanical Turk[76]. Otras son más específicas de pruebas, como UTest[77]. También se encuentran propuestas gratuitas en ámbitos universitarios [184].

4.2.4.6. Análisis del nivel de mejora 4 y los problemas en Pruebas Continuas

En el nivel de mejora 4 se implementan prácticas como selección de pruebas, paralelización, uso de API para ejecutar precondiciones, rotación de navegadores, entre otras. Este nivel no solo implica aprovechar mejor el tiempo en la ejecución de las pruebas, sino que también busca incorporar mecanismos de mejora continua para las mismas. El conducto de Prueba Continua de este nivel se muestra en la Fig. 60.

Este nivel, soluciona los problemas de pruebas que consumen mucho tiempo de ejecución que han sido reportados por la literatura y la industria. En total, en este nivel CTIL, se obtiene un 61,11% de mejora en estos problemas (11 problemas solucionados de 18), que con el 11,11% logrado en el nivel 1, el 22,22% del nivel 2 y el 5,56% obtenido en el nivel 3, se alcanza el 100%.

[76] https://www.mturk.com/
[77] https://www.utest.com/

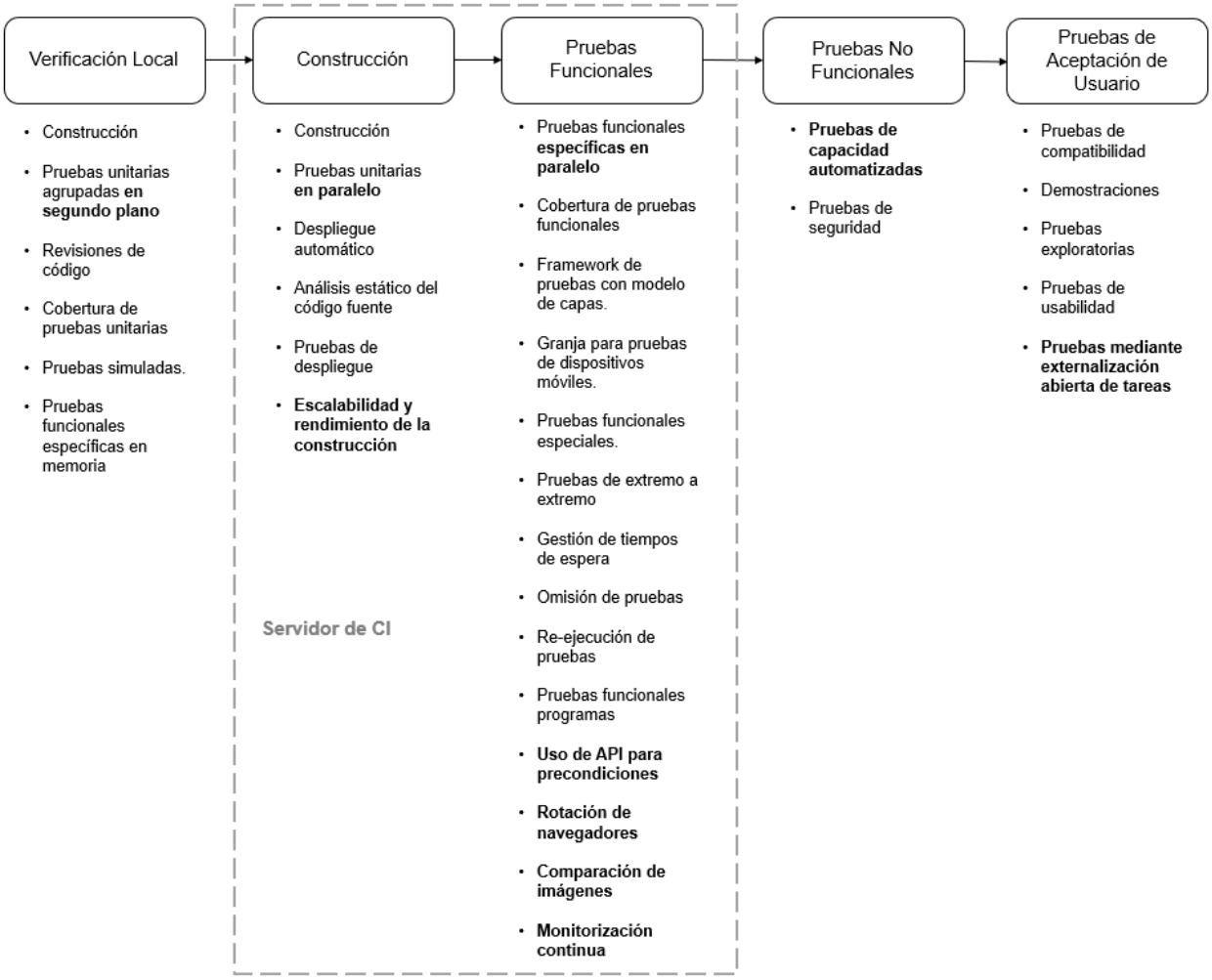


Fig. 60. Conducto de Pruebas Continuas en el nivel CTIL 4.

Las mejoras de este CTIL se presentan en la Fig. 61.

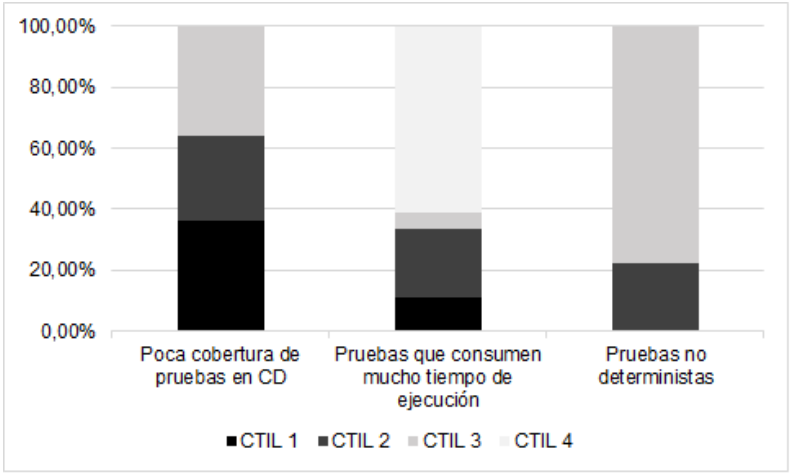


Fig. 61. Grado de resolución de los problemas de Prueba Continua al implementar el nivel 4 del modelo.

4.2.5. Relación entre los niveles de mejora y los problemas relacionados a las Pruebas Continuas

La Tabla 55 resume el modelo propuesto y lista todas las buenas prácticas que lo componen.

Tabla 55. Buenas prácticas del modelo de mejoras para Pruebas Continuas.

Nivel	Etapa	Buena Práctica
1	Verificación Local	1.1.1. Pruebas unitarias automatizadas.
		1.1.2. Revisiones de código.
	Construcción	1.2.1. Construcción automática.
		1.2.2. Ejecución automática de pruebas unitarias.
	Pruebas funcionales	1.3.1. Pruebas funcionales automatizadas.
		1.3.2. Cobertura de pruebas funcionales.
	Pruebas no funcionales	1.4.1. Medición de tiempos de respuesta.
	Pruebas de aceptación de usuario	1.5.1. Pruebas de compatibilidad web.
		1.5.2. Demonstraciones a interesados en el sistema.
2	Verificación Local	2.1.1. Agrupamiento de pruebas unitarias.
		2.1.2. Cobertura de pruebas unitarias.
	Construcción	2.2.1. Despliegue automático.
	Pruebas funcionales	2.3.1. Implementación de un modelo de capas para el marco de trabajo de pruebas funcionales.
		2.3.2. Segmentación de pruebas funcionales.
		2.3.3. Granja para pruebas de dispositivos móviles.
	Pruebas no funcionales	2.4.1. Gestión de pruebas de capacidad.
	Pruebas de aceptación de usuario	2.5.1. Implementación y gestión de pruebas exploratorias.
3	Verificación Local	3.1.1. Uso de pruebas simuladas.
		3.1.2. Ejecución de pruebas funcionales en memoria.

Nivel	Etapa	Buena Práctica
	Construcción	3.2.1. Análisis estático del código fuente.
		3.2.2. Pruebas de despliegue e instalación.
	Pruebas funcionales	3.3.1. Automatización de pruebas especiales.
		3.3.2. Automatización de pruebas de extremo a extremo.
		3.3.3. Gestión de asincronía y tiempos de espera.
		3.3.4. Estrategia de omisión de pruebas.
		3.3.5. Sistema de re-ejecución de pruebas fallidas.
		3.3.6. Ejecución de pruebas funcionales programadas.
		3.3.7. Sistema de generación y eliminación de datos de pruebas.
		3.3.8. Generación de reportes precisos.
	Pruebas no funcionales	3.4.1. Gestión de pruebas de seguridad.
	Pruebas de aceptación de usuario	3.5.1. Pruebas de usabilidad.
4	Verificación Local	4.1.1. Pruebas unitarias en segundo plano.
	Construcción	4.2.1. Pruebas unitarias en paralelo.
		4.2.2. Escalabilidad y rendimiento de la construcción.
	Pruebas funcionales	4.3.1. Pruebas funcionales en paralelo.
		4.3.2. Selección de pruebas a ejecutar.
		4.3.3. Uso de API para la ejecución de precondiciones.
		4.3.4. Rotación de navegadores.
		4.3.5. Comparación de imágenes.
		4.3.6. Monitorización continua de pruebas funcionales.
	Pruebas no funcionales	4.4.1. Automatización de pruebas de capacidad.
	Pruebas de aceptación de usuario	4.5.1. Pruebas mediante externalización abierta de tareas.

Cada buena práctica presentada en los diferentes niveles CTIL forman parte de la solución a cada grupo de problema de Pruebas Continua (poca cobertura de pruebas en Entrega Continua, pruebas que consumen mucho tiempo de ejecución y pruebas no deterministas).

Como resumen de este modelo, en la Tabla 56 se presenta como cada uno de estos tres problemas es abordado por las diferentes prácticas del modelo CTIM.

Tabla 56. Relación entre las buenas prácticas del modelo CTIM y los problemas en Pruebas Continua.

Problema en Pruebas continuas	**CTIM**		**Etapa en el conducto de Pruebas continuas**
	Nivel	**Buena práctica**	
Poca cobertura de pruebas en Entrega Continua	1	1.1.1. Pruebas unitarias automatizadas.	Verificación Local
		1.1.2. Revisiones de código.	
		1.2.1. Construcción automática.	Construcción
		1.2.2. Ejecución automática de pruebas unitarias.	
		1.3.1. Pruebas funcionales automatizadas.	Pruebas funcionales
		1.3.2. Cobertura de pruebas funcionales.	
		1.4.1. Medición de tiempos de respuesta.	Pruebas no funcionales
		1.5.1. Pruebas de compatibilidad web.	Pruebas de aceptación de usuario
		1.5.2. Demonstraciones a interesados en el sistema.	
	2	2.1.1. Agrupamiento de pruebas unitarias.	Verificación Local
		2.1.2. Cobertura de pruebas unitarias.	
		2.3.1. Implementar un modelo de capas para el marco de trabajo de pruebas funcionales.	Pruebas funcionales
		2.3.2. Segmentación de pruebas funcionales.	
		2.3.3. Granja para pruebas de dispositivos móviles.	
		2.4.1. Gestión de pruebas de capacidad.	Pruebas no funcionales
		2.5.1. Implementación y gestión de pruebas exploratorias.	Pruebas de aceptación de usuario

Problema en Pruebas continuas	CTIM		Etapa en el conducto de Pruebas continuas
	Nivel	Buena práctica	
	3	3.1.1. Uso de pruebas simuladas.	Verificación Local
		3.1.2. Ejecución de pruebas en memoria.	
		3.2.1. Análisis estático del código fuente.	Construcción
		3.3.1. Automatización de pruebas especiales.	Pruebas funcionales
		3.3.2. Automatización de pruebas de extremo a extremo.	
		3.3.3. Gestión de asincronía y tiempos de espera.	
		3.3.8. Generación de reportes ambiguos.	
		3.4.1. Gestión de pruebas de seguridad.	Pruebas no funcionales
		3.5.1. Pruebas de usabilidad.	Pruebas de aceptación de usuario
Pruebas que consumen mucho tiempo de ejecución	1	1.1.1. Pruebas unitarias automatizadas.	Verificación Local
		1.3.1. Pruebas funcionales automatizadas.	Pruebas funcionales
	2	2.1.1. Agrupamiento de pruebas unitarias.	Verificación Local
		2.1.2. Cobertura de pruebas unitarias.	
		2.2.1. Despliegue automático.	Construcción
		2.3.2. Segmentación de pruebas funcionales.	Pruebas funcionales
	3	3.3.1. Automatización de pruebas especiales.	Pruebas funcionales
	4	4.1.1. Pruebas unitarias en segundo plano.	Verificación Local
		4.2.1. Pruebas unitarias en paralelo.	Construcción
		4.2.2. Escalabilidad y rendimiento de la construcción.	
		4.3.1. Pruebas funcionales en paralelo.	Pruebas funcionales
		4.3.2. Selección de pruebas a ejecutar.	
		4.3.3. Uso de API para la ejecución de precondiciones.	
		4.3.4. Rotación de navegadores.	

Problema en Pruebas continuas	CTIM		Etapa en el conducto de Pruebas continuas
	Nivel	Buena práctica	
		4.3.5. Comparación de imágenes.	
		4.3.6. Monitorización continua de pruebas funcionales.	
		4.4.1. Automatización de pruebas de capacidad.	Pruebas no funcionales
		4.5.1. Pruebas mediante externalización abierta de tareas.	Pruebas de aceptación de usuario
Pruebas no deterministas	2	2.3.1. Implementar un modelo de capas para el marco de trabajo de pruebas funcionales.	Pruebas funcionales
		2.4.1. Gestión de pruebas de capacidad.	Pruebas no funcionales
	3	3.2.2. Pruebas de despliegue e instalación.	Construcción
		3.3.1. Automatización de pruebas especiales.	Pruebas funcionales
		3.3.3. Gestión de asincronía y tiempos de espera.	
		3.3.4. Estrategia de omisión de pruebas.	
		3.3.5. Sistema de re-ejecución de pruebas fallidas.	
		3.3.6. Ejecución de pruebas funcionales programadas.	
		3.3.7. Sistema de generación y eliminación de datos de pruebas.	

5. PROCESO DE EVALUACIÓN CON EL MODELO CTIM

Generalmente, cuando una organización considera que ha mejorado sus procesos con respecto a un determinado nivel de CMMI, es normal que decida presentarse a una auditoría que ratifique dicha mejora a través de un tercero certificado por el Instituto de Ingeniería del Software (SEI). Del mismo modo, el modelo de mejoras para Pruebas Continuas está dirigido a organizaciones que pretendan validar el cumplimiento de buenas prácticas de acuerdo a los diferentes niveles y etapas del mismo.

En esta sección, se describe el proceso de evaluación con el modelo CTIM, incluyendo el mecanismo de evaluación, con cada una de sus etapas y la herramienta desarrollada para evaluar un nivel CTIL utilizando CTIM.

5.1. Proceso de evaluación

El proceso de evaluación se basa en la misma metodología que utiliza el método SCAMPI [185]. SCAMPI son las siglas de Método de Evaluación CMMI Estándar para la Mejora de Proceso (Standard CMMI Appraisal Method for Process Improvement). Es una metodología de evaluación oficial para determinar la calificación de madurez CMMI de una organización determinada. Esta metodología consta de tres etapas y son:

- Preparación y planificación de la auditoría: en esta etapa se definen los objetivos de la mejora y se selecciona el método de captura de evidencias. Esta tarea la realiza el auditor externo o responsable de la auditoría.
- Revisión de la preparación: durante esta actividad se analiza si la organización está preparada para la evaluación.
- Ejecución de la auditoría y reporte de resultados: en esta etapa, todos los miembros del equipo evaluador ejecutan la evaluación final de un nivel de mejora de CTIM y comunican los resultados posteriormente.

5.1.1. Unidad organizacional y proyectos auditados

De forma similar a cuando se va a evaluar un nivel de madurez CMMI por medio de una auditoría SCAMPI, el subconjunto o parte de la organización que será evaluada se llama “unidad

organizacional". Una vez definida, se determina el grupo de proyectos que van a ser evaluados del total de la misma. Este grupo se denomina "muestra de proyectos".

La unidad organizacional es la parte de la organización que va a ser auditada. Los departamentos o áreas que componen la unidad organizacional deberán tener procesos coherentes y objetivos de negocio comunes. En caso de una empresa pequeña, la unidad organizacional puede ser toda la organización.

Por ejemplo, una empresa multinacional con sedes en Estados Unidos, México y España puede definir como unidad organizacional a una de las sedes. Asimismo, puede darse el caso de una empresa con áreas bien definidas (como por ejemplo desarrollo del sitio web para usuarios finales, desarrollo del sistema interno para empleados, etc.) y cuya operación también esté distribuida en varias sedes geográficamente, pero que decida que la unidad organizacional sea una de estas áreas, aunque involucre a varias sedes.

Cuando se va a realizar una auditoría CMMI, normalmente no se evalúan todos los proyectos de la unidad organizacional, sino que se selecciona una muestra representativa [186]. La elección de los proyectos para la muestra es una tarea importante dentro del proceso de evaluación de CTIM, ya que estos deben cubrir todos los factores críticos identificados dentro de la unidad organizacional.

5.1.2. Participantes y etapas de la auditoría

Al igual que en CMMI, para llevar a cabo una auditoría CTIM es necesario un equipo de trabajo que cumpla unos requisitos mínimos relacionados a experiencia, conocimiento y habilidades [185]. Dentro de este equipo de trabajo se identifican los siguientes roles:

- Espónsor: es el responsable de liderar el proyecto de mejora.
- Auditores externos: son los encargados de realizar la evaluación. Deben ser externos a la organización que se está auditando.
- Responsables internos: forman parte de la organización a auditar y cuentan con experiencia suficiente para participar de la misma. Generalmente, son representados por líderes técnicos, responsables de área o jefes técnicos de proyectos.
- Participantes seleccionados: personal encargado de brindar información necesaria para la auditoría.

Como se mencionó en la sección 5.1, la primera etapa es la preparación y planificación de la auditoría. Durante la etapa de preparación, se determinan los objetivos de la mejora y se selecciona el método de captura de evidencias. Esta tarea la lleva a cabo el auditor externo. Luego, los miembros de la organización a ser evaluada normalmente recopilan y organizan evidencia objetiva documentada según en el alcance de los artefactos disponibles dentro del proyecto y alineadas con el alcance de la evaluación. La planificación comienza con la comprensión de los objetivos, requisitos y limitaciones del patrocinador de la evaluación. Es fundamental determinar el alcance de la evaluación, es decir, el CTIL objetivo y las etapas de V&V que se desean evaluar.

La preparación previa tanto del equipo de evaluación como del proyecto a ser evaluado es clave para la ejecución más eficiente del método. El análisis de la evidencia objetiva documentada preliminar proporcionada por el proyecto es importante para la evaluación. Si faltan datos sustanciales en este punto, las actividades de evaluación posteriores pueden retrasarse o incluso cancelarse.

La recolección previa de documentación objetiva por parte de los miembros del proyecto a ser evaluado impacta positivamente durante la etapa de evaluación, incluyendo beneficios como clarificaciones de resultados, detalles de implementaciones, miembros involucrados en diferentes actividades como parte de una buena práctica, etc. Sin embargo, el esfuerzo de recopilar, organizar y revisar grandes cantidades de evidencia objetiva antes de la evaluación significa un costo para el proyecto a ser evaluado y puede conducir a una disminución del rendimiento si no se hace de manera eficiente. Hay tres tipos básicos de evidencia enumerados en la documentación de SCAMPI, los cuales también se utilizan para este proceso de evaluación. La Tabla 57 lista los tipos de evidencia.

Tabla 57. Tipos de evidencia.

Tipo de evidencia	Descripción	Ejemplos
Artefactos directos	Los productos tangibles que resultan directamente de la implementación de una buena práctica de Pruebas Continuas. Una parte integral de la verificación de la implementación de la práctica. Puede estar explícitamente establecido o implícito el procedimiento, documentación o material informativo asociado.	- Resultados enumerados en las prácticas. - Enlaces a archivos, repositorios, etc. - Documentos, productos entregables, materiales de capacitación, etc.

Tipo de evidencia	Descripción	Ejemplos
Artefactos indirectos	Artefactos que son consecuencia de la realización de otra buena práctica o que no son parte de la implementación de la práctica que se está evaluando. Este tipo de indicador es especialmente útil cuando existen dudas sobre si se ha cumplido la intención de la práctica (por ejemplo, existe un artefacto, pero no hay indicación de cómo surgió, quién trabajó para desarrollarlo o cómo se usa).	- Resultados enumerados en las prácticas. - Minutas de reuniones, revisión de resultados, informes de estado, presentaciones, etc. - Medidas de desempeño.
Afirmaciones	Declaraciones orales o escritas que confirmen o respalden la implementación (o falta de implementación) de una buena práctica. Estas declaraciones generalmente son proporcionadas por los implementadores de la práctica, responsables de los proyectos de desarrollo y/o clientes internos o externos. También pueden ser mencionadas por otras partes interesadas (por ejemplo, gerentes y dueños de producto).	- Instrumentos. - Entrevistas. - Cuestionarios. - Presentaciones, demonstraciones, etc.

Posteriormente, se ejecuta la etapa de revisión de la preparación. El objetivo de esta tarea es conocer si la organización a auditar está preparada para afrontar la evaluación. Durante esta etapa, se analizan las evidencias objetivas, las experiencias del equipo de trabajo y la disposición logística (equipamiento, instalaciones, disponibilidad, etc.) para validar si es factible la realización de la auditoría.

Utilizando las evidencias, los auditores externos ejecutan la evaluación utilizando la herramienta descrita en la sección 5.2. Una vez que todas las etapas de V&V objetivos fueron calificadas, la herramienta muestra de forma automática los resultados de la evaluación. Con estos resultados, el equipo evaluador provee reportes a las partes interesadas. Finalmente, se elabora el plan de acción correctivo, que incluye las acciones a realizar para mejorar o implementar las buenas prácticas que no fueron calificadas como satisfactorias.

5.2. Descripción de EvalCTIM

La herramienta de evaluación del modelo CTIM, denominada EvalCTIM (Fig. 62), fue desarrollada utilizando Java, JSP, HTML y Bootstrap.

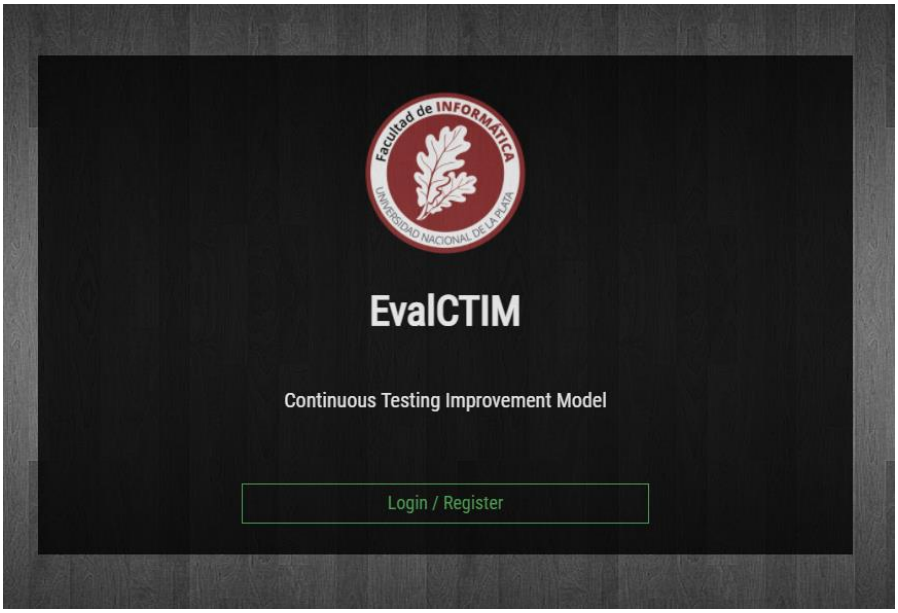


Fig. 62. Herramienta de evaluación utilizada

Es un sitio web que permite a un usuario evaluador crearse una cuenta, introducir los datos de la organización y realizar evaluaciones (Fig. 63).

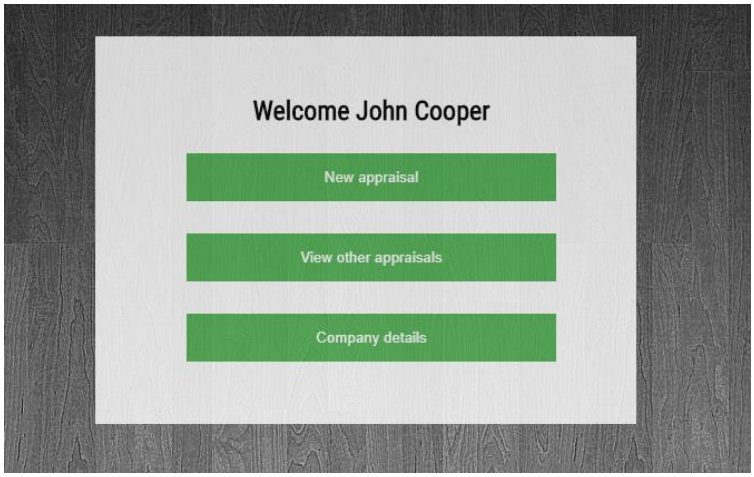


Fig. 63. Menú principal del evaluador.

Cuando el usuario selecciona la opción de crear una nueva evaluación, se solicita al mismo un nombre como identificador de la evaluación, los espónsores, una breve descripción del propósito de la evaluación, procesos involucrados y el alcance de la evaluación, es decir, etapas de V&V a evaluar y el nivel de CTIM objetivo. Esto se muestra en la Fig. 64. La información requerida es la misma que se solicita a los evaluadores en otras herramientas que evalúan CMMI, como por ejemplo Appraisal Assitant[78].

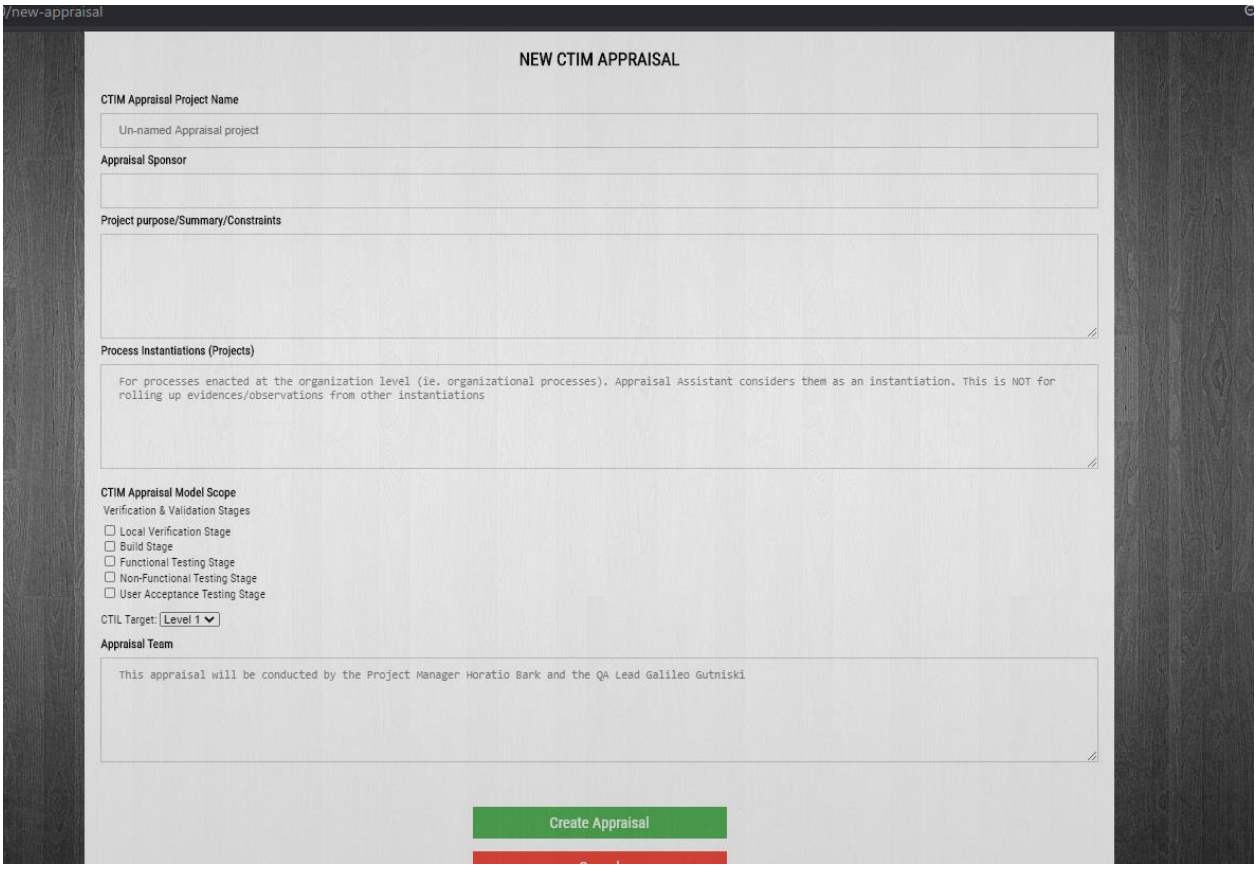


Fig. 64. Pantalla para la generación de una nueva evaluación.

Una vez iniciada la evaluación, las pantallas siguientes corresponden a cada etapa de V&V dentro del CTIL que fue seleccionado como objetivo. Por cada pantalla, se presentan las diferentes buenas prácticas de Pruebas Continuas que propone el modelo (Fig. 65), que al ser expandidas muestran el objetivo y descripción de la misma y una pregunta que ayuda al evaluador a determinar si se satisface o no (Fig. 66).

[78] https://appraisal-assistant.software.informer.com/

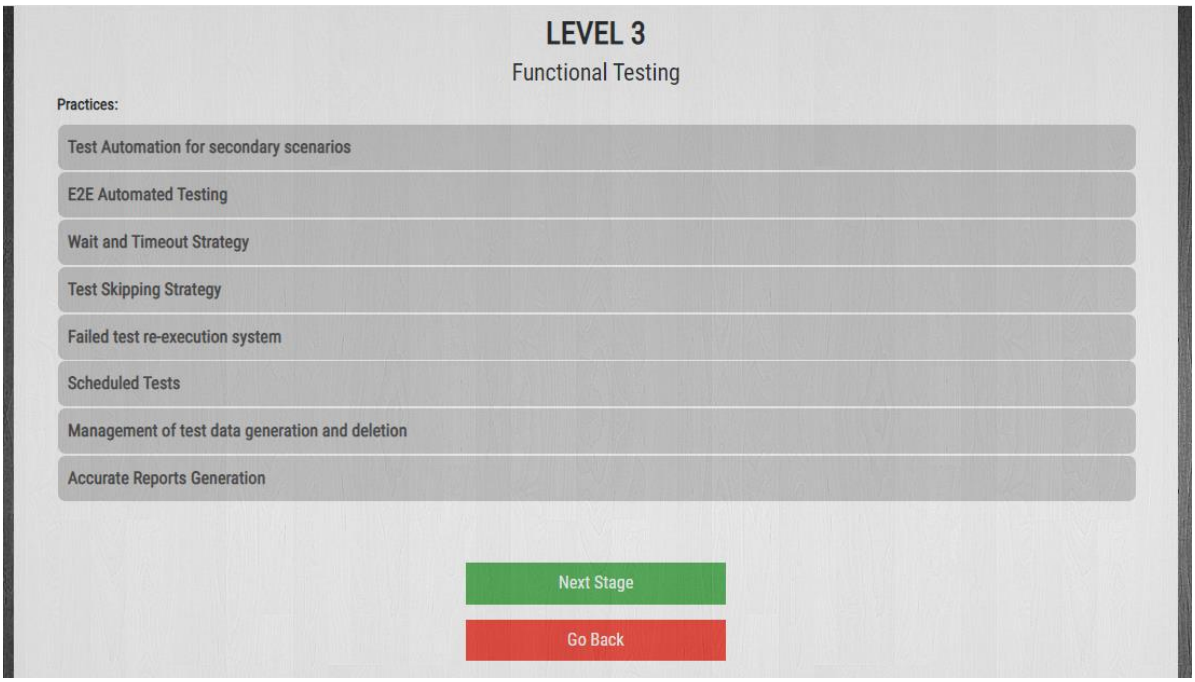


Fig. 65. Pantalla de evaluación para una etapa de V&V del nivel CTIL objetivo.

Fig. 66. Práctica de Pruebas Continuas en la herramienta.

Además de las opciones para indicar que la práctica es satisfactoria o no, la herramienta ofrece otras alternativas. Si la práctica no aplica al proyecto (por ejemplo, una práctica para pruebas sobre aplicaciones nativas en dispositivos móviles cuando el proyecto no realiza ese desarrollo), la herramienta provee una opción para indicarlo. Finalmente, si el evaluador considera que no se tienen suficientes evidencias para calificar la implementación de la práctica, la herramienta brinda una opción de “sin calificar” (not rated).

Finalmente, cuando la evaluación finaliza, se muestra la página de resultados. La misma muestra un resumen de la evaluación de cada etapa de V&V, un gráfico que indica el porcentaje de progreso para cada una de ellas, para alcanzar el nivel CTIL objetivo y por último el Conducto de Pruebas Continuas obtenido (ver Fig. 67). Adicionalmente, se detalla el resultado de cada práctica y se listan las evidencias y observaciones que se realizaron a lo largo de la evaluación (ver Fig. 68).

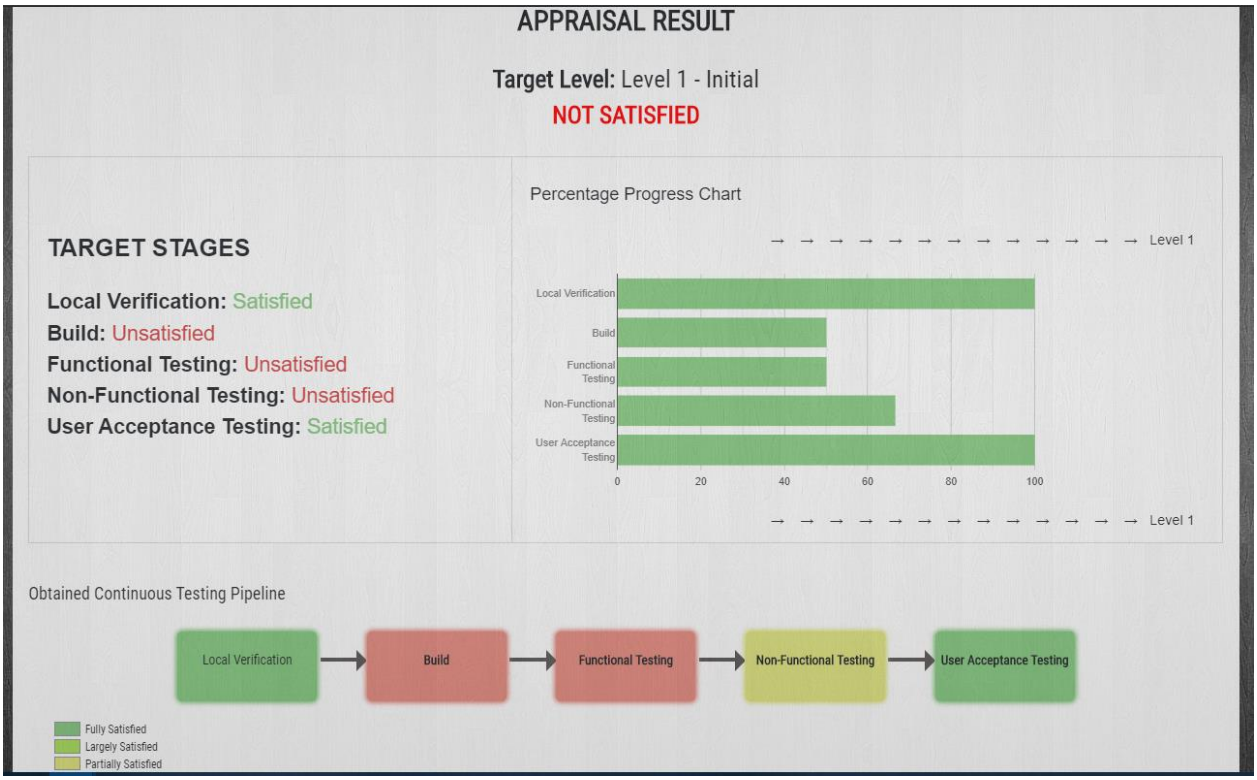


Fig. 67. Página de resultado de la evaluación (primera parte)

Fig. 68. Página de resultado de la evaluación (segunda parte)

La página de los resultados también contiene un enlace a una página donde se muestran acciones correctivas sugeridas por la herramienta, de acuerdo a la evaluación realizada. Esto se muestra en la Fig. 69. Estas acciones correctivas, utiliza como base el formato AENOR[79].

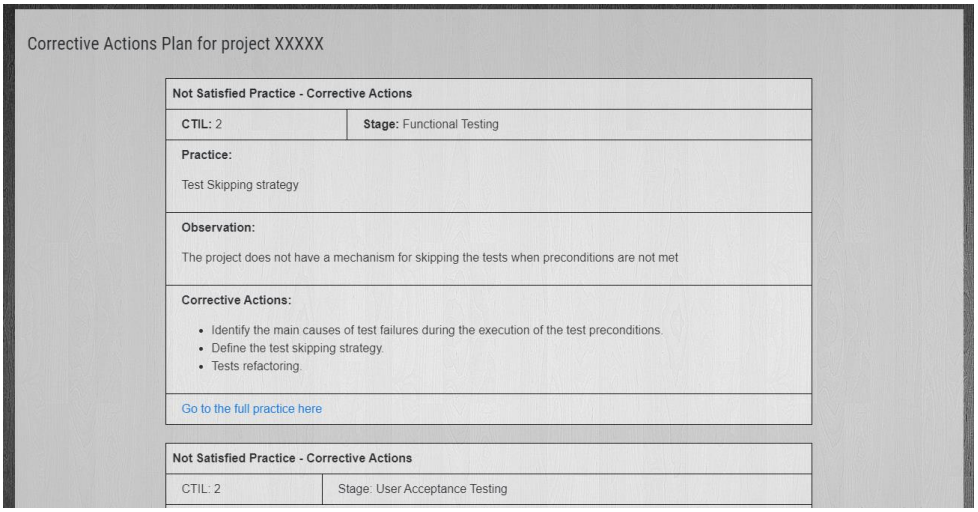


Fig. 69. Acciones correctivas en la herramienta.

[79] https://www.aenor.com/

6. VALIDACIÓN DEL MODELO CTIM

En este apartado se describe la validación realizada al modelo CTIM detallado anteriormente. Las mismas se han abordado siguiendo el método de Investigación-Acción (IA) descrito en la sección 1.4.2 de este trabajo.

En el marco del método de IA, un proceso de validación está compuesto por grupos de actividades organizadas formando un ciclo característico. Las actividades principales de cada ciclo son:

- Planificación: identificar las cuestiones de investigación a llevarse a cabo en el ciclo, relacionadas con la mejora del modelo CTIM.
- Acción: el modelo CTIM es aplicado en condiciones reales de uso, o analizado por expertos en el área.
- Observación: recolección de los datos obtenidos al aplicar el modelo CTIM.
- Reflexión: análisis de los datos obtenidos, para mejorar el modelo CTIM realizando un registro del mismo.

El proceso de validación utilizado para validar el modelo CTIM se divide en dos etapas de validación bien diferenciadas (Fig. 70). La entrada es el modelo propuesto y la salida es el modelo validado.

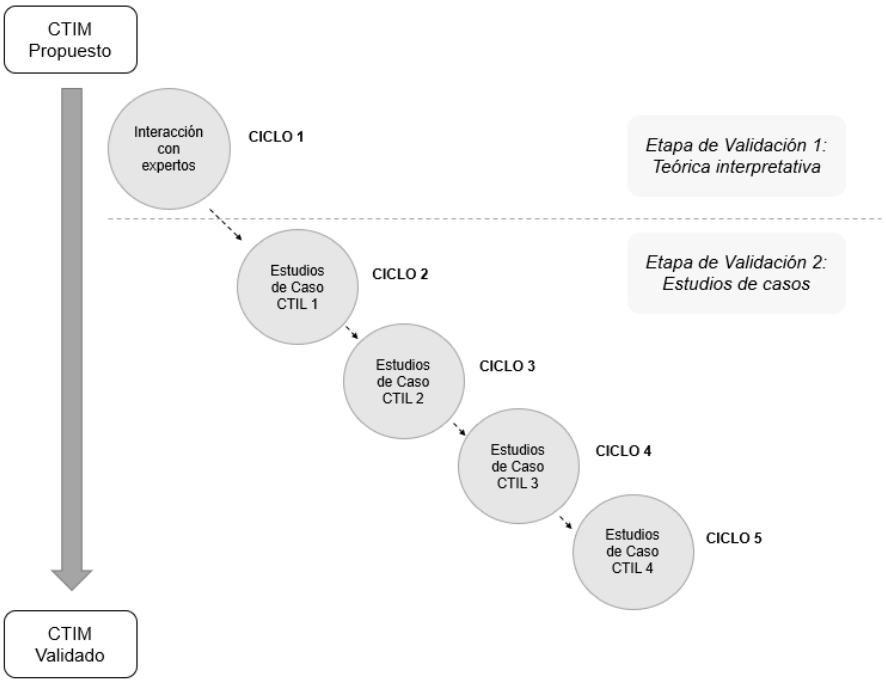


Fig. 70. Ciclos de IA para la validación del modelo CTIM

La primera etapa de validación consiste en una perspectiva teórica interpretativa. Esta perspectiva conduce directamente a un método cualitativo, implicando una interacción con personas que experimentan directamente los problemas y soluciones investigados. En la segunda etapa, cada ciclo de IA consiste en la evaluación de un proyecto de desarrollo de software real y el análisis de la implementación de los diferentes niveles CTIL. Esto se conoce como “refinamiento” dentro de IA y también contribuye a la introducción de mejoras en el modelo CTIM. Estos ciclos de investigación formados por la evaluación e implementación del modelo se realizarán mediante estudios de casos.

Los roles que conforman el método de IA son: el investigador (doctorando), los expertos y profesionales seleccionados (GCR), el objeto Investigado (modelo CTIM) y los beneficiarios. Los beneficiarios son aquellos proyectos de desarrollo de software pertenecientes a diferentes empresas. La relación entre los distintos roles se ilustra en la Fig. 71. Finalmente, los métodos de selección de los expertos participantes se detallan en las secciones siguientes.

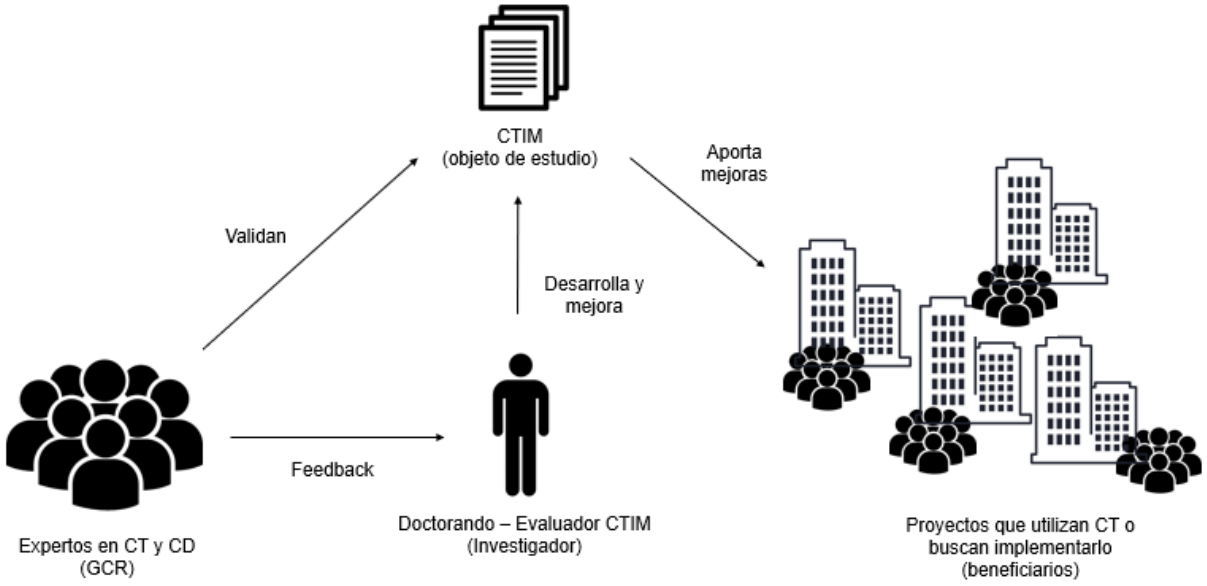


Fig. 71. Participantes en la Investigación-Acción.

6.1. Etapa de validación 1: Interpretación teórica del modelo

En esta primera etapa, se realiza una validación del modelo, desde una perspectiva teórica. Este tipo de enfoque ha demostrado ser eficaz en la validación de marcos de trabajos y modelos para la mejora de procesos de software [187]–[189].

Este ciclo de IA ha sido planificado para ejecutarse en cuatro fases:

- Identificación y selección de expertos participantes y diseño de la entrevista.
- Evaluación teórica del modelo.
- Entrevistas semi-estructuradas con los participantes.
- Análisis de los resultados y elaboración de conclusiones para la mejora del modelo.

6.1.1. Selección de los expertos participantes

Es necesaria una selección de expertos cuidadosa para evitar sesgos, incertidumbres e información incompleta [190]. Para hacerlo, se utilizó el modelo de Fehring [191] en combinación con la propuesta de Herranz et al. [189] para dar pesos a ciertos criterios relacionados con la experiencia profesional y académica. Estos criterios con sus pesos se presentan en la Tabla 58.

Tabla 58. Criterios y pesos para la selección de expertos auditores.

ID	Criterio	Peso
C1	10 o más años de experiencia en calidad de software y desarrollo continuo.	4
C2	Experiencia docente en calidad de software y desarrollo continuo, en instituciones destacadas.	3
C3	PhD en el campo de calidad de software y desarrollo continuo.	3
C4	Disertaciones en el campo de la calidad de software y desarrollo continuo.	2
C5	Maestría en el campo de calidad de software y desarrollo continuo.	2
C6	Publicación de artículos científicos relacionados a PC en revistas internacionales.	2
C7	Certificados y/o especializaciones en PC otorgados por instituciones destacadas.	1
C8	Publicación de artículos científicos relacionados a PC en workshops.	1

Para que los profesionales fueran considerados expertos en Pruebas Continuas o Entrega Continua y seleccionados para participar en los estudios, se requiere una puntuación de 7, de acuerdo con Herranz et al. [189]. De esta forma se obtuvo una lista de expertos en Pruebas Continuas aplicando el criterio de Fehring (Tabla 58), analizando sus perfiles de LinkedIn, CV en sitios web personales cuando era posible y sus perfiles en medios académicos como Scopus, ORCID o Google Scholar.

Luego de realizar la búsqueda, se encontraron 32 expertos con un promedio mayor a 7 puntos, de los cuales aceptaron participar 17. Los participantes con su respectiva puntuación se muestran en la Tabla 59, ordenados por números del 1 al 17.

Tabla 59. Características de los expertos evaluadores del ciclo 1 de IA.

Experto	C1	C2	C3	C4	C5	C6	C7	C8	Puntuación
1	X			X				X	7
2	X	X		X	X	X		X	14
3	X	X	X	X	X	X		X	17
4	X			X			X	X	8
5	X	X		X	X	X		X	14
6	X	X		X	X	X		X	14
7		X		X	X	X	X	X	11
8	X		X	X		X		X	12
9	X	X		X	X	X		X	14
10	X			X			X	X	8
11	X	X							7
12	X			X				X	7
13	X	X		X			X	X	11
14	X	X	X	X	X	X	X	X	18
15	X	X							7
16	X			X				X	7
17	X	X		X	X	X		X	14
N	16	11	3	15	8	9	6	15	190
(%)	95%	65%	18%	89%	48%	53%	30%	89%	**11**

Como se puede observar en la Tabla 59, los expertos obtuvieron un promedio de 11 puntos, donde el 95% de los mismos tienen 10 o más años de experiencia en calidad de software y desarrollo continuo, pero una notoria falta de formación académica relacionada, especialmente doctorados (18%).

6.1.2. Diseño de la entrevista

Como siguiente paso se diseñó una encuesta con preguntas cerradas, semi-cerradas y abiertas, a realizarse como parte de una entrevista semi-estructurada en línea.

Las preguntas fueron las siguientes:

- Indique en una escala del 1 al 5, si el modelo significa una solución para los problemas de Prueba o Entrega Continua que enfrenta la industria en la actualidad.
- Indique en una escala del 1 al 5, la viabilidad de implementación del modelo en un equipo de desarrollo de software.
- Indique en una escala del 1 al 5, que tan sencilla considera la implementación del modelo.
- ¿Renombraría algún nivel del modelo?
- ¿Qué niveles consideraría eliminar del modelo?
- ¿Agregaría un nivel al modelo? En caso de que sí, ¿cuál sería el alcance del mismo?
- ¿Renombraría alguna etapa del modelo?
- ¿Qué etapas consideraría eliminar del modelo?
- ¿Qué etapa agregaría y en qué nivel?
- ¿Qué prácticas agregaría, en qué etapa y en qué nivel?
- ¿Qué prácticas consideraría que puedan ser eliminadas?
- ¿Hay alguna problemática de Prueba Continua que no haya sido considerada por el modelo?
- ¿Tiene alguna sugerencia para mejorar el modelo?

Con estas preguntas, se llevaron a cabo entrevistas luego de la evaluación teórica del modelo que se describe seguidamente.

6.1.3. Evaluación teórica del modelo

Para la evaluación del modelo, se brindó a cada participante un resumen estructural del modelo, conteniendo los diferentes niveles, etapas y prácticas. Además, se presentó una imagen descriptiva del mismo similar a la Fig. 41, junto con su estructura (Tabla 55). Por último, el modelo completo se anexó como documento de guía.

A medida que un experto confirmaba la finalización de la evaluación, se lo invitaba a participar de las entrevistas. El proceso total de evaluación y entrevistas tardó cuatro meses. Los resultados de las entrevistas se presentan a continuación.

Los primeros atributos evaluados tienen que ver con la percepción de los expertos respecto a si el modelo representa una solución para los problemas de Pruebas Continuas en la industria. Además, la viabilidad y simplicidad de su implementación en las empresas. Estos datos se muestran en la Fig. 72.

Todos los expertos coinciden en que el modelo significa una solución para Pruebas Continuas y la mayoría de ellos (82%) está totalmente de acuerdo con ello. Respecto a la factibilidad de implementación, el mismo porcentaje coincide con que la viabilidad es muy alta. Sin embargo, solo 2 de los expertos afirman que es una tarea sencilla.

Respecto a niveles y etapas por renombrar, solo se hicieron las sugerencias mencionadas en la Tabla 60 para las etapas. En la segunda columna se indica entre paréntesis el número de sugerencias recibidas.

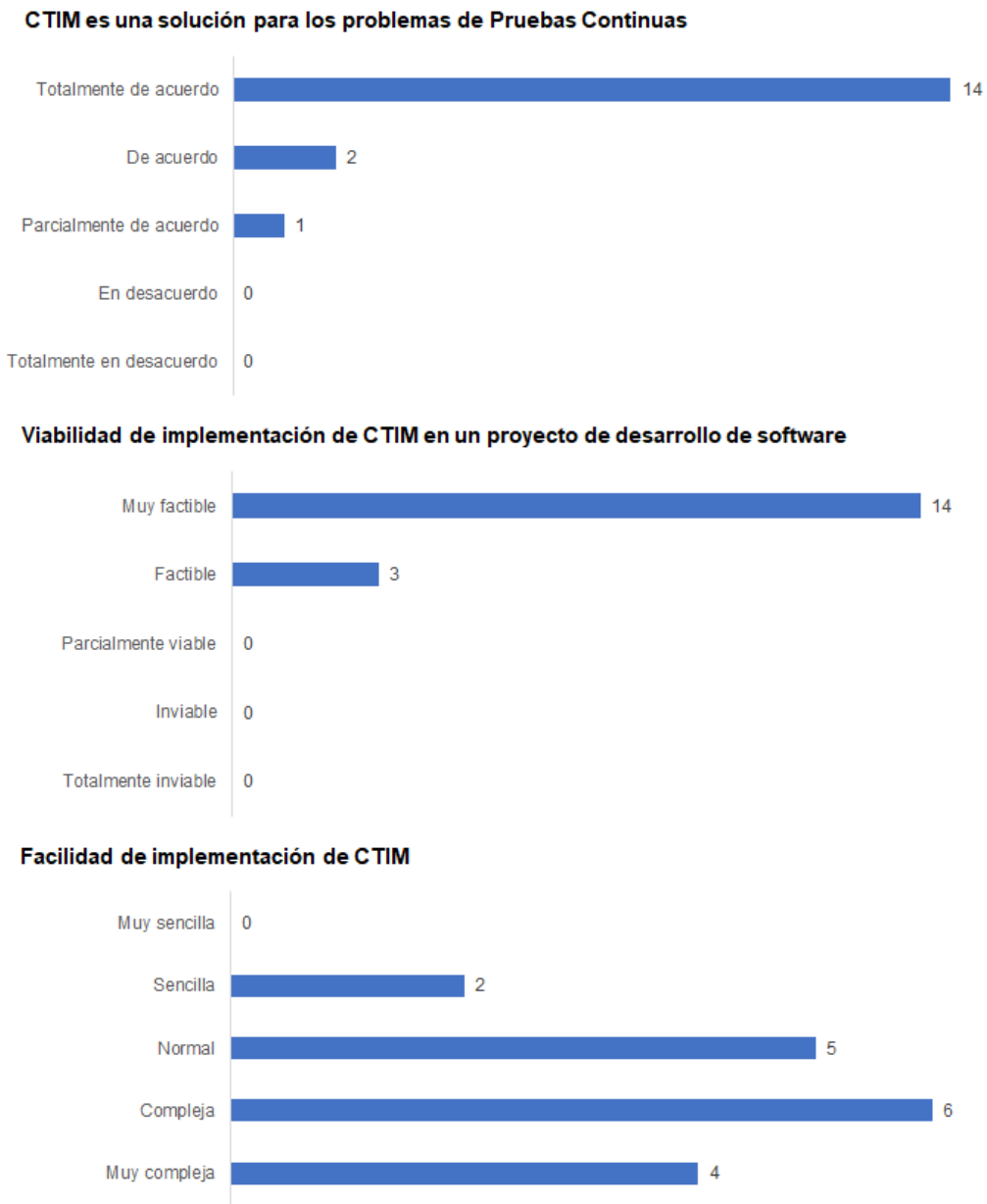


Fig. 72. Opiniones de los expertos respecto al modelo CTIM.

Tabla 60. Sugerencias para renombrar etapas de V&V del modelo.

Etapa de V&V Actual	Etapa de V&V sugerida
Etapa de pre-compilación	• Etapa de pre-commit (1) • Etapa de pruebas en ambiente local (2) • Etapa de verificación local (3) • Etapa local (2)
Etapa de pruebas de aceptación	• Etapa de pruebas de sistema (1) • Etapa de pruebas funcionales (7)
Etapa de pruebas de aceptación de usuario	• Etapa de pruebas manuales (2)

Las etapas de pre-compilación y pruebas de aceptación fueron las que recibieron más sugerencias respecto al cambio de nombre. En menor medida, la etapa de pruebas de aceptación de usuario recibió 2 sugerencias para ser renombrada a etapa de “pruebas manuales”. Respecto a la eliminación de niveles y etapas del modelo, ninguno de los autores sugirió hacerlo. En este mismo contexto, no se mencionaron recomendaciones relacionadas con la adición de nuevos niveles y etapas.

Por el lado de las prácticas los expertos sugirieron agregar 7 más al modelo. Estas se listan en la Tabla 61.

Tabla 61. Prácticas sugeridas por los expertos para ser agregadas al modelo.

Práctica sugerida	Etapa de V&V	CTIL
Pruebas funcionales en memoria.	Pre-compilación	3
Pruebas de integración.	Pruebas de aceptación	3
Pruebas de extremo a extremo.	Pruebas de aceptación	3
Pruebas de usabilidad.	Pruebas de aceptación de usuario	3
Pruebas de seguridad.	Pruebas no funcionales	3
Generación del ambiente para pruebas en aplicaciones móviles.	Pruebas de aceptación	2
Planificación de pruebas de rendimiento (umbrales, tipos, entorno, etc.)	Pruebas no funcionales	2

Del mismo modo que los niveles y las etapas, los evaluadores no sugieren quitar ninguna práctica del modelo.

Respecto a la cuestión acerca de identificar si hay alguna problemática de Pruebas Continuas que no haya sido considerada por el modelo, los expertos mencionaron que no se presentaron soluciones para las pruebas en Big Data.

Finalmente, se obtuvieron las siguientes mejoras respecto de las preguntas sobre el agregado de algún nivel más al modelo y sugerencias adicionales que contribuyan a la mejora del mismo:

- El modelo es muy práctico y puede contribuir muy positivamente a la mejora del proceso de pruebas de una organización, en un entorno continuo. Sin embargo, su implementación no parece ser simple. Se sugiere que al igual que otros modelos de referencia o estándares como CMMI, la implementación se realice con la supervisión de un experto.
- El modelo carece de ilustraciones gráficas y ejemplos de algoritmos para la implementación de prácticas. Se recomienda añadirlos todas las veces que sea posible.
- El modelo debe contemplar la mejora continua, es decir, permitir la generación de diferentes versiones constantemente, ya que los problemas en las pruebas de software evolucionan y varían con el correr del tiempo. Asimismo, cada año se publican nuevas propuestas en la literatura que podrían ser agregadas posteriormente.
- El modelo no contempla pruebas de seguridad ni pruebas de usabilidad. Estas dos son muy importantes para muchos proyectos y deberían ser agregadas al modelo, abarcando todos sus tipos y aspectos.

6.1.4. Análisis de los resultados y mejora del modelo

A partir de los resultados obtenidos se elaboraron mejoras para el modelo CTIM. Estas ya se encuentran reflejadas en la versión final del modelo presentado en la sección 4, junto con las otras mejoras que resultaron en los ciclos de validación posteriores.

En primer lugar, analizando la opinión de los expertos se puede confirmar la validez del modelo propuesto como solución a los problemas de Pruebas Continuas en las empresas. Esto es una conclusión importante ya que permite someter al modelo a un proceso de mejoras mínimas posteriores, en lugar de rehacerlo con otro enfoque.

De este modo, en base al resto de las sugerencias y recomendaciones de los expertos se implementaron mejoras resumidas en la Tabla 62.

Tabla 62. Mejoras implementadas en el modelo CTIM luego del primer ciclo de IA.

Alcance	Mejora
CTIM	Se añade en la definición del modelo, que el mismo debe ser implementado en un proyecto bajo la supervisión de un experto.
Etapa V&V	Modificación del nombre de la etapa **Pre-compilación** por **Verificación Local**
Etapa V&V	Modificación del nombre de la etapa **Pruebas de aceptación** por **Pruebas funcionales**
Práctica	Se agrega la práctica **ejecución de pruebas funcionales en memoria**, en la etapa de **Verificación Local**, del nivel CTIL 3.
Práctica	Se agrega la práctica **automatización de pruebas de extremo a extremo**, en la etapa de **Pruebas funcionales**, del nivel CTIL 3.
Práctica	Se agrega la práctica **pruebas de usabilidad**, en la etapa de **Pruebas de aceptación de usuario**, del nivel CTIL 3.
Práctica	Se agrega la práctica **gestión de pruebas de seguridad**, en la etapa de **pruebas no funcionales**, del nivel CTIL 3.
Práctica	Se agrega recomendaciones relacionadas a la detección de vulnerabilidades de seguridad, en la práctica **análisis estático del código fuente**, del CTIL 3.
Práctica	Se agrega la práctica **granja para pruebas de dispositivos móviles**, en la etapa de **pruebas funcionales**, del CTIL 2.
Práctica	Se modifica la práctica **pruebas de capacidad** por **gestión de pruebas de capacidad**, en la etapa de **pruebas no funcionales**, del CTIL 2.
CTIM	Se añaden ilustraciones y algoritmos.

Asimismo, algunas recomendaciones no se implementaron. Estas junto con la justificación se listan a continuación.

- El nombre de la etapa “Pruebas de aceptación de usuario” no fue modificado a “Pruebas manuales”. La razón es que esta etapa abarca más que un proceso llevado a cabo de forma manual por un equipo especializado. Contempla demostraciones al cliente, evaluaciones automatizadas de aspectos de las pruebas orientados al usuario final, etc. Es por ello que este mismo nombre lo utilizan los autores de Entrega Continua [7].
- La práctica de pruebas de integración no se agrega al modelo, puesto que las mismas son contempladas en otros tipos de pruebas según el contexto y la naturaleza del proyecto donde se las está implementando.

- Las pruebas de Big Data no se han incorporado como parte del alcance de esta versión del modelo (sección 1.5), debido a la carencia de buenas prácticas tanto en la industria como en la literatura académica [11], [14].

Posteriormente, con el modelo validado por los expertos, se inicia la segunda etapa de validación, la cual se describe en el siguiente apartado.

6.2. Etapa de validación 2: Estudios de casos

El objetivo de cada estudio de caso llevado a cabo es analizar la implementación del modelo CTIM en diferentes proyectos de desarrollo de software, tomando uno por cada estudio. Estas tareas son ejecutadas por responsables internos de cada proyecto y un par de auditores externos seleccionados, guiados por el doctorando. Para alcanzar este objetivo, cada estudio de caso se planifica del mismo modo, a través cuatro fases consecutivas:

- Fase 1: realización de una evaluación utilizando la herramienta descrita en el apartado 5.2 y obtención de un Plan de Acciones Correctivas (PAC) compuesto por buenas prácticas para el nivel CTIL objetivo.
- Fase 2: implementación de estas buenas prácticas en el proyecto evaluado.
- Fase 3: recolección de datos acerca de las fases 1 y 2, mediante entrevistas con los responsables internos y auditores externos.
- Fase 4: análisis de la información recopilada para identificar mejoras al modelo CTIM por cada práctica.

El método de selección tanto de los proyectos y auditores participantes como de la entrevista utilizada se describen a continuación.

6.2.1. Selección de los proyectos participantes

Para elegir los proyectos que participarán en los estudios de caso, se buscaron responsables del área de control de calidad o jefes de proyecto con experiencia de más de 10 años utilizando metodologías ágiles y en entornos de desarrollo continuo. Estos años de experiencia se tomaron considerando las recomendaciones de Freimut, Briand y Vollei [190]. La búsqueda se realizó utilizando la red social LinkedIn.

De este modo, a partir del criterio mencionado se obtuvo una lista de 18 candidatos. Todos fueron contactados por correo electrónico para participar en la investigación, pero solo 10 aceptaron. La propuesta consistía en formar pares de auditores entre los elegidos para:

- Evaluar el proceso de pruebas de un proyecto de desarrollo de software. El mismo pertenece a un participante distinto del par y se utiliza la herramienta descrita en la sección 5.2.
- Analizar los PAC generados por la herramienta EvalCTIM.
- Colaborar en la implementación de las prácticas de CTIM.

Una vez finalizada la implementación, los mismos serían entrevistados para retroalimentar el modelo. El beneficio que los mismos obtendrían a cambio, sería el soporte y colaboración por parte de otros dos profesionales para la evaluación e implementación de Pruebas Continuas en su proyecto.

De este modo, para cada estudio de caso participaron junto con el doctorando, un responsable interno y dos auditores externos. El responsable interno es el jefe de proyecto o responsable del área de control de calidad del proyecto en cuestión. Tanto el responsable interno como los auditores externos son miembros de los 10 participantes seleccionados. Esta relación entre evaluadores se presenta en la Tabla 63. Los proyectos participantes pertenecen a las empresas mencionadas en la sección 1.4.2, pero para mantener la confidencialidad respecto al proyecto en sí y los auditores, solo se los nombra utilizando letras (proyecto A, proyecto B, etc.)

Tabla 63. Matriz de responsables internos y auditores externos para cada estudio de caso.

Caso	Responsable Interno	Auditor Externo 1	Auditor Externo 2
Proyecto A	Profesional Proyecto A	Profesional Proyecto B	Profesional Proyecto C
Proyecto B	Profesional Proyecto B	Profesional Proyecto C	Profesional Proyecto D
Proyecto C	Profesional Proyecto C	Profesional Proyecto D	Profesional Proyecto E
Proyecto D	Profesional Proyecto D	Profesional Proyecto E	Profesional Proyecto F
Proyecto E	Profesional Proyecto E	Profesional Proyecto F	Profesional Proyecto G
Proyecto F	Profesional Proyecto F	Profesional Proyecto G	Profesional Proyecto H
Proyecto G	Profesional Proyecto G	Profesional Proyecto H	Profesional Proyecto I
Proyecto H	Profesional Proyecto H	Profesional Proyecto I	Profesional Proyecto J
Proyecto I	Profesional Proyecto I	Profesional Proyecto J	Profesional Proyecto A
Proyecto J	Profesional Proyecto J	Profesional Proyecto A	Profesional Proyecto B

En total, se llevaron a cabo diez implementaciones del modelo CTIM en los diferentes proyectos, pero solo se describen cuatro estudios de caso (uno por cada CTIL). Es importante resaltar que si bien todos los estudios iniciaron al mismo tiempo, la prioridad de implementación de los PAC se realiza en función del nivel CTIL obtenido: primero se ejecutaron los estudios de caso correspondientes al nivel CTIL 1 y los últimos en llevarse a cabo fueron los pertenecientes al nivel CTIL 4. De esta manera, el proceso de validación de la etapa 2, se encuentra alineado con los ciclos representados en la Fig. 70.

6.2.2. Entrevistas

El propósito de las entrevistas es recopilar información acerca de la aplicación del modelo CTIM por los evaluadores internos y externos. Para hacerlo, por cada estudio de caso se llevaron a cabo entrevistas estructuradas utilizando cuestionarios en línea, donde los expertos pudieron responder rápidamente a las preguntas. El cuestionario estaba formado por las siguientes preguntas:

- **Pregunta 1**: ¿Aplican los tipos de pruebas del proyecto a las etapas de V&V del modelo?
- **Pregunta 2**: ¿Son interpretadas correctamente cada práctica del modelo para la etapa de V&V y el nivel CTIL en evaluación?
- **Pregunta 3**: ¿Faltan pasos y/o tareas adicionales para la implementación de cada práctica?
- **Pregunta 4**: ¿Hay pasos y/o tareas redundantes o innecesarios en la implementación de cada práctica?
- **Pregunta 5**: ¿La implementación de cada práctica asegura la solución parcial de un problema de Pruebas Continuas?

La pregunta 1 busca asegurar que todos los tipos de pruebas y niveles de pruebas ejecutadas por el proyecto evaluado son contemplados dentro del modelo CTIM como parte de las etapas de V&V. La pregunta 2 tiene como objetivo verificar que las prácticas propuestas y las tareas, sean comprendidas por los evaluadores y evitar así ambigüedades. Las preguntas 3 y 4 buscan identificar inconsistencias en las prácticas, sobre pasos y tareas que deban ser agregadas o eliminadas. Por último, la pregunta 5 busca confirmar que la práctica propuesta por el modelo CTIM según el nivel CTIL y la etapa de V&V, una vez implementada, aporte a la resolución de problemas en las Pruebas Continuas.

6.2.3. Ciclo 2: Estudio de caso para CTIL 1

El estudio de caso se realizó sobre una plataforma web y móvil de una compañía de compras de viajes en línea. La misma comercializa pasajes de vuelos, hoteles, cruceros y alquiler de autos. El proyecto consta de tres equipos de trabajo, formado por 18 desarrolladores, 6 especialistas en pruebas y 1 jefe de proyecto. El proyecto realiza despliegues a producción una vez por mes y su objetivo es acelerar este proceso a una vez cada dos semanas. La metodología de desarrollo utilizada es la Programación Extrema (XP).

Como parte del objetivo de aumentar la velocidad de los despliegues a producción, el proyecto encuentra la necesidad de comenzar a implementar prácticas del desarrollo continuo de software.

6.2.3.1. Fase 1: Evaluación del proceso de pruebas

Como parte de la fase 1 se procede a realizar la evaluación del modelo CTIM utilizando la herramienta descrita en la sección 5.2. Previamente, los auditores externos con la colaboración del responsable interno recopilan la mayor cantidad de evidencias posibles. Se recolectaron enlaces a repositorios del código fuente (código base, de las pruebas unitarias y de las pruebas funcionales), enlaces a solicitudes de integración de código previamente generados, artefactos generados por el servidor de Integración Continua, reportes de ejecución de pruebas, citas a reuniones en el calendario del equipo y documentos generados por la herramienta de gestión de casos de pruebas.

Con toda la evidencia obtenida, los auditores externos llevan a cabo la evaluación y el resultado obtenido para el nivel CTIL 1 es **no satisfactorio**. El reporte con los resultados de la evaluación se muestra en la Fig. 73 y los detalles en la Fig. 74.

Los resultados indican que existen problemas a resolver en las etapas de V&V de “pruebas funcionales”, “pruebas no funcionales” y “pruebas de aceptación de usuario”. En primer lugar, en la etapa de pruebas funcionales, los casos de pruebas no se etiquetan cuando son automatizados, lo cual dificulta la medición de la cobertura.

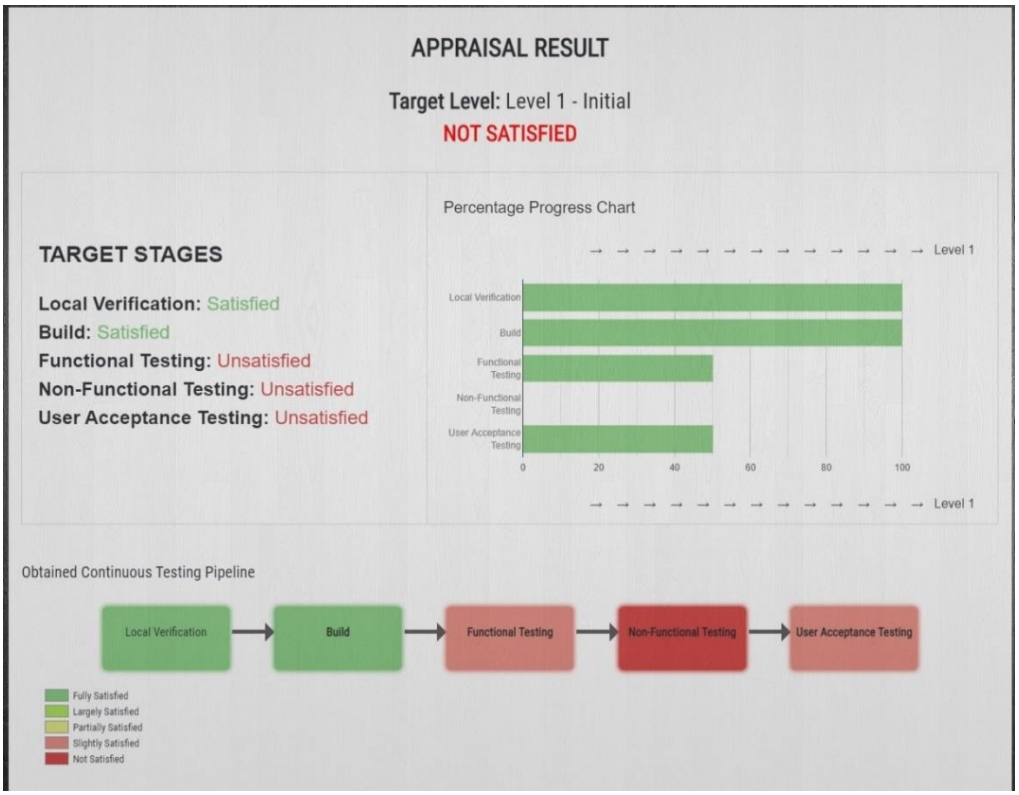
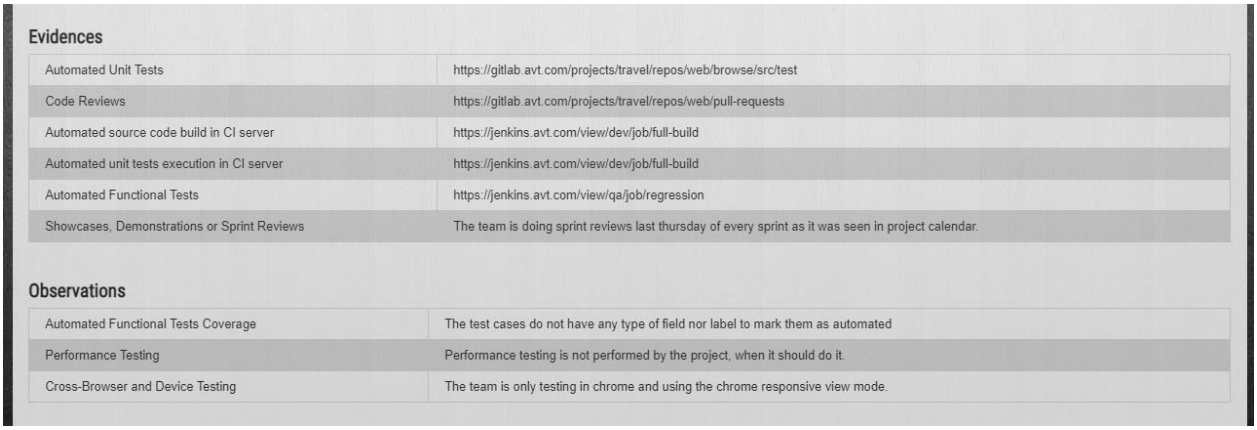


Fig. 73. Resultado de la evaluación del estudio de caso 1.

Fig. 74. Detalles de los resultados de la evaluación del estudio de caso 1.

En lo que respecta a las pruebas no funcionales, faltan mecanismos para medir el rendimiento del sitio en desarrollo. Por el lado de las pruebas de aceptación de usuario, el equipo de pruebas manuales no está ejecutando pruebas en todos los navegadores y dispositivos

soportados. Todas estas observaciones, en conjunto con las evidencias de las prácticas que fueron satisfactorias se muestran en la Fig. 75.



Evidences

Automated Unit Tests	https://gitlab.avt.com/projects/travel/repos/web/browse/src/test
Code Reviews	https://gitlab.avt.com/projects/travel/repos/web/pull-requests
Automated source code build in CI server	https://jenkins.avt.com/view/dev/job/full-build
Automated unit tests execution in CI server	https://jenkins.avt.com/view/dev/job/full-build
Automated Functional Tests	https://jenkins.avt.com/view/qa/job/regression
Showcases, Demonstrations or Sprint Reviews	The team is doing sprint reviews last thursday of every sprint as it was seen in project calendar.

Observations

Automated Functional Tests Coverage	The test cases do not have any type of field nor label to mark them as automated
Performance Testing	Performance testing is not performed by the project, when it should do it.
Cross-Browser and Device Testing	The team is only testing in chrome and using the chrome responsive view mode.

Fig. 75. Evidencias y observaciones para el estudio de caso CTIL 1.

Finalmente, la Tabla 64 muestra las acciones correctivas propuestas por el PAC generado por en la herramienta.

Tabla 64. Acciones correctivas para el estudio de caso 1.

Práctica	Etapa	CTIL	Observaciones	Acciones correctivas
Cobertura de pruebas funcionales	Pruebas funcionales	1	Los casos de pruebas no tienen ningún tipo de campo o sistema de etiquetado para marcarlos como “automatizados”.	• Incorporación de un campo adicional en los casos de pruebas para marcarlos como “automatizados” o “no automatizados”. • Revisar los casos de pruebas ya automatizados para marcarlos como “automatizados”.
Pruebas de capacidad	Pruebas no funcionales	1	No se ejecuta ningún tipo de prueba de capacidad.	• Selección de una herramienta para realizar pruebas de rendimiento. • Medición de tiempos de respuesta, para la detección de anomalías en ellas.

Práctica	Etapa	CTIL	Observaciones	Acciones correctivas
Pruebas de compatibilidad web.	Pruebas de aceptación de usuario	1	El equipo de pruebas solo realiza verificaciones en el navegador “Google Chrome” y su modo de diseño adaptable.	• Incorporar en el plan de pruebas la ejecución de pruebas de aceptación manuales en todos los navegadores y dispositivos soportados. • Estimar el esfuerzo de hacerlo como parte de cada requerimiento.

6.2.3.2. Fase 2: Implementación del PAC

En primer lugar, se solicitó al equipo de gestión de la configuración que añadan un campo a los casos de pruebas para identificar si los mismos están automatizados o no. El equipo de gestión de la configuración está a cargo de la administración de los servidores y herramientas que utilizan los diferentes proyectos de desarrollo. Esta sugerencia fue realizada por uno de los auditores externos. La solicitud se completó en tan solo 1 día. Posteriormente, se introdujo un requerimiento dentro del proyecto para la revisión de todos los casos existentes, con el objetivo de identificar los que estaban automatizados. Esto llevó 15 días de trabajo, en paralelo con la ejecución de otras actividades del proyecto que no podían ser pasadas por alto.

Luego, se decidió planificar la ejecución de pruebas de capacidad. Antes de comenzar la etapa de implementación, se destinó 1 semana de trabajo a realizar pruebas de concepto con diferentes herramientas existentes en el mercado. Las características recomendadas por los expertos para determinar la selección fueron:

- Licencia.
- Facilidad de uso.
- Escalabilidad.
- Reportes claros.

Una vez finalizado el análisis de las pruebas de concepto, el responsable interno optó por utilizar la herramienta JMeter. Solicitó a los especialistas en pruebas que se familiaricen y capaciten en el uso de la herramienta, mediante videotutoriales y documentación. A la par, el equipo de desarrollo fue construyendo un plan de ejecución de pruebas de capacidad, donde el

objetivo era medir los tiempos de respuesta del sitio en desarrollo. Este proceso de capacitación y diseño del plan de ejecución de pruebas de capacidad llevó 2 semanas. Por último, se añadió al alcance de cada requerimiento, la ejecución de estas pruebas con cada despliegue al ambiente previo a producción.

Finalmente, se pusieron en marcha las acciones correctivas para la ejecución de pruebas de aceptación manuales en todos los navegadores y dispositivos soportados. Uno de los desafíos que se encontró en este punto, fue el de detectar cuales son los navegadores y dispositivos más utilizados por los usuarios finales del sitio de la agencia de viajes. Por esta razón, los expertos recomendaron el uso de la herramienta “Google Analytics” para obtener esta información. La implementación de la herramienta se realizó en 1 día, pero se optó por esperar 1 mes para que la herramienta obtenga datos representativos. Transcurrido el mes, la herramienta contenía información acerca de los navegadores y dispositivos más usados:

- Google Chrome (68%)
    - Windows (48%)
    - OSx (12%)
    - Linux (2%)
    - Android (36%)
    - iOS (2%)
- Safari (21%)
    - OSx (73%)
    - iOS (27%)
- Microsoft Edge (5%)
- Internet Explorer (4%)
- Mozilla Firefox (2%)

La lista de navegadores y dispositivos más usados se añadió al plan de pruebas. Los expertos sugirieron realizar una reunión con los dueños del producto para proponer ampliar el alcance de las pruebas a más de un navegador y dispositivo. La fundamentación es asegurar que la calidad del sitio web sea la misma para los usuarios que utilicen navegadores diferentes a Google Chrome. Sin embargo, el tiempo destinado a la ejecución de pruebas aumentaría considerablemente, por lo que, durante la reunión entre los expertos, los dueños de producto y el responsable del proyecto, se determinó solo realizar pruebas en aquellos navegadores que son usados por el 5% o más de los usuarios (Google Chrome, Safari y Microsoft Edge). Esta información también fue incluida en el plan de pruebas. A partir de ese momento, se comenzó a

sumar el esfuerzo de la ejecución de pruebas en esos navegadores incorporados para cada requerimiento.

6.2.3.3. Fase 3: Recolección de datos

Para la validación de este nivel CTIL del modelo, se invitó a los auditores externos y al responsable interno, a participar de las entrevistas mencionadas en la sección 6.2.2. Los resultados de las entrevistas se presentan a continuación.

- **Pregunta 1:** ¿Aplican los tipos de pruebas del proyecto a las etapas de V&V del modelo? Los tres encuestados, respondieron que sí.
- **Pregunta 2*:*** ¿Son interpretadas correctamente cada práctica del modelo para la etapa de V&V y el nivel CTIL en evaluación? Uno de los expertos respondió satisfactoriamente. Sin embargo, el responsable interno junto con el otro experto, sugirieron la siguiente recomendación:
    - No todos los casos de pruebas pueden ser automatizados. Habría que contemplar la posibilidad de detectar la cobertura de los casos de pruebas automatizados sobre los que solo pueden ser automatizados, en lugar de hacerlo sobre la totalidad de los casos. Esto podría clarificarse en la práctica de cobertura de pruebas automatizadas.
- **Pregunta 3*:*** ¿Faltan pasos y/o tareas adicionales para la implementación de cada práctica? En las respuestas a esta pregunta, se identificaron las siguientes recomendaciones:
    - Los dos auditores externos detectaron un problema al momento de determinar los navegadores y dispositivos utilizados por los usuarios finales del sitio web de la agencia de viaje. Por ello, recomiendan la incorporación de una serie pasos previos en la práctica de pruebas de compatibilidad web: implementar una herramienta de analítica web como Google Analytics, obtener los navegadores y dispositivos de los usuarios y añadir esta información al plan de pruebas.
    - El responsable interno notó que los especialistas en pruebas a veces olvidaban realizar las pruebas en un navegador o dispositivo. Por ello, sugiere que se añada una tarea final en las pruebas Cross-Browser que sea solicitar a quien ejecuta la prueba que indique en el requerimiento los navegadores y dispositivos donde se

hicieron las verificaciones. Otra alternativa podría ser una lista de verificación con la lista de navegadores y dispositivos por cada requerimiento.

- **Pregunta 4:** ¿Hay pasos y/o tareas redundantes o innecesarios en la implementación de cada práctica? Nuevamente, en esta pregunta, todos los participantes no consideraron mejoras por hacer.
- **Pregunta 5:** ¿La implementación de cada práctica asegura la solución parcial de un problema de Pruebas Continuas? Los expertos respondieron que "si bien la evaluación se hizo en un nivel inicial de mejora, las prácticas sugeridas por el modelo en esta instancia son las bases para enfocarse más adelante en otros factores de Entrega Continua como la velocidad y la confiabilidad".

6.2.3.4. Fase 4: Análisis de los resultados y mejora del modelo

A partir de las respuestas obtenidas en las entrevistas realizadas, se llevó a cabo un análisis que concluyó con la inclusión de mejoras en el modelo CTIM. Como se mencionó anteriormente, estas ya se encuentran reflejadas en la versión final del modelo presentado en la sección 4, junto con el resto de las mejoras.

En la Tabla 65, se muestran las mejoras realizadas al modelo en este estudio de caso, como parte del segundo ciclo de IA.

Tabla 65. Mejoras implementadas para el nivel CTIL 1 luego del segundo ciclo de IA.

Etapa de V&V	Mejora
Pruebas funcionales	En la práctica de **cobertura de pruebas automatizadas**, se incorpora una aclaración para calcular la cobertura teniendo en cuenta solo los casos de pruebas automatizables.
Pruebas de aceptación de usuario	Se añade la tarea **Implementación de herramienta de analítica web** a la práctica de **Pruebas de compatibilidad web**.
Pruebas de aceptación de usuario	Se añade la tarea **Obtención de métricas de navegadores y dispositivos más utilizados** a la práctica de **Pruebas de compatibilidad web**.

Etapa de V&V	Mejora
Pruebas de aceptación de usuario	Se modifica la práctica **Pruebas de compatibilidad web**, incorporando al final una sugerencia de una lista de verificación. La misma lista los navegadores y dispositivos por orden de prioridad, para que no sean olvidados durante la ejecución de las pruebas.

6.2.4. Ciclo 3: Estudio de caso para el nivel CTIL 2

El segundo estudio de caso se realizó sobre una plataforma web de contenido sobre educación y ciencia, con temáticas en las ciencias naturales, geografía, medio ambiente, biología, arqueología, la historia y patrimonio histórico. La empresa que la desarrolla es una de las organizaciones más grandes del mundo con contenido educativo y científico. El proyecto que fue evaluado consta de 6 equipos ágiles, formado por 6 desarrolladores, 1 especialista en pruebas automatizadas y 1 especialista en pruebas manuales. Todos los equipos trabajan bajo la supervisión de un líder técnico y un jefe de proyecto.

Los equipos ya se encontraban desarrollando pruebas automatizadas en diferentes niveles (unitarias, de integración, funcionales y no funcionales), que estaban programadas para ejecutarse diariamente por la noche.

Al momento de realizar el estudio de caso, uno de los objetivos principales del proyecto era la introducción de Integración Continua para detectar errores con la ejecución de pruebas automatizadas tan pronto como sea posible, es decir, al momento de integrar los cambios al repositorio de control de versiones. De este modo, el líder técnico decidió participar de la experimentación del modelo.

6.2.4.1. Fase 1: Evaluación del proceso de pruebas

Como se mencionó en la sección 6.2, inicia la fase 1 con la evaluación del modelo CTIM utilizando la herramienta. Antes de iniciar, se destinó una semana a la recolección de evidencias directas e indirectas, coordinada por el líder técnico y el jefe de proyecto, siguiendo las recomendaciones del doctorando.

Al finalizar la etapa de recolección de evidencias, los auditores externos llevaron a cabo la evaluación, con objetivo en el nivel CTIL 1. Como se muestra en la Fig. 76, el resultado fue **satisfactorio**.

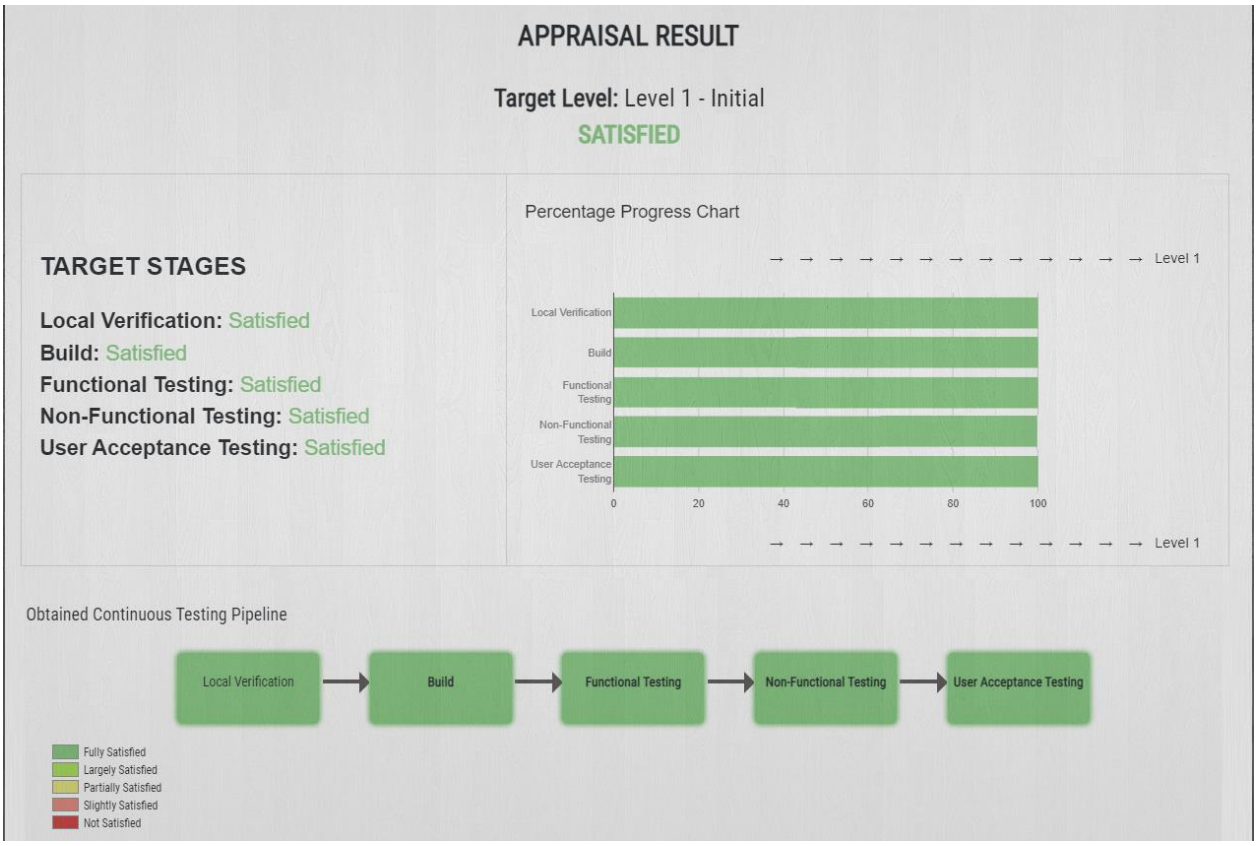


Fig. 76. Resultado de la primera evaluación del estudio de caso 2.

Inmediatamente después, se realiza otra evaluación, configurando el nivel CTIL 2 como objetivo. Esta vez, el resultado fue **no satisfactorio**. El reporte con los resultados de la evaluación se muestra en la Fig. 77.

Los resultados de la evaluación señalan problemas en las etapas de V&V de “verificación local”, “construcción, “pruebas funcionales” y “pruebas no funcionales”. En primer lugar, en la etapa de verificación local, no se está midiendo la cobertura de pruebas unitarias. En segundo lugar, en la etapa de construcción, el despliegue no está ejecuta de forma automática después de la construcción del código. los casos de pruebas no se etiquetan cuando son automatizados, lo cual dificulta la medición de la cobertura. Por el lado de las pruebas funcionales automatizadas, las mismas no están agrupadas por todos los criterios recomendados, solo por prioridad. Finalmente, dentro de las pruebas no funcionales, si bien se miden los tiempos de respuesta con cada despliegue del sistema, no se tienen umbrales para determinar éxito o fallo. Estas observaciones junto con las respectivas acciones correctivas propuestas por el PAC generado se muestran en la Tabla 66.

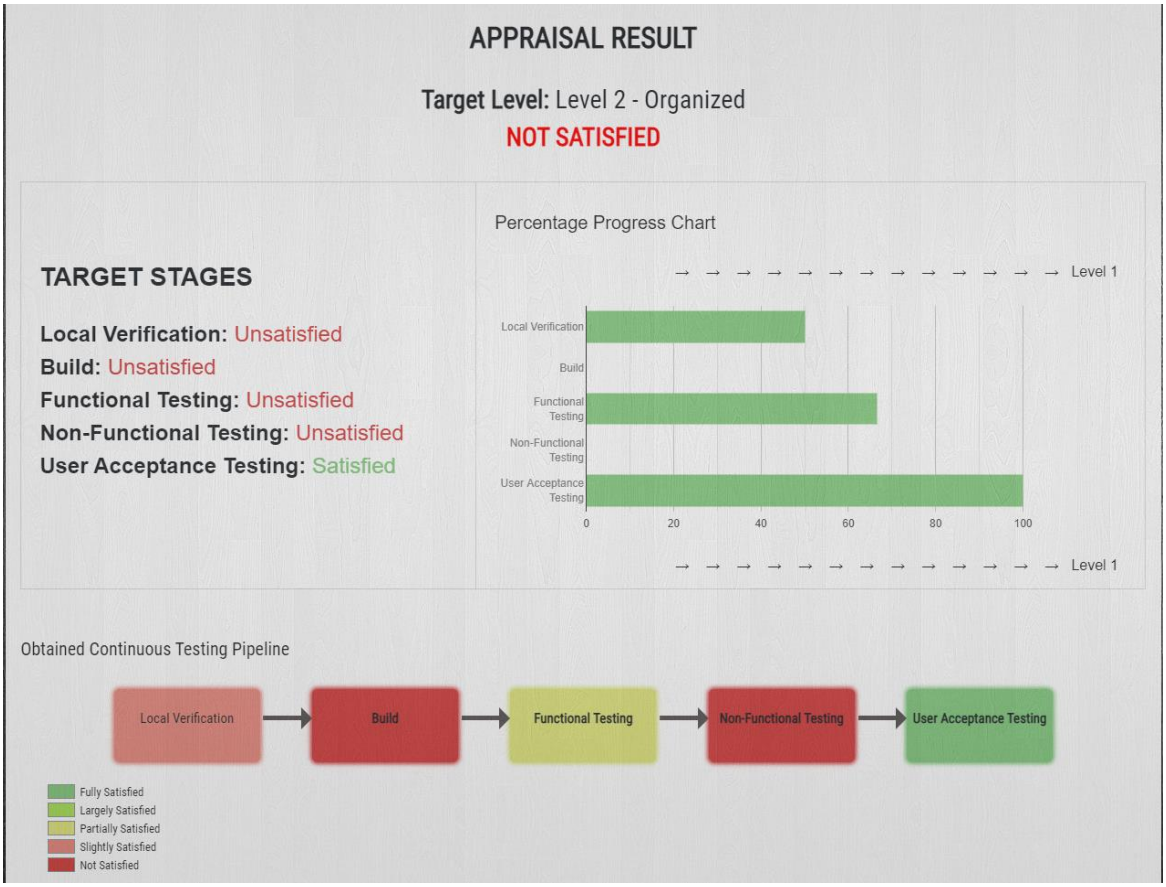


Fig. 77. Resultado de la segunda evaluación del estudio de caso 2.

Tabla 66. Acciones correctivas para el estudio de caso 2.

Práctica	Etapa	CTIL	Observaciones	Acciones correctivas
Cobertura de pruebas unitarias	Verificación local	2	No se mide la cobertura de pruebas unitarias.	• Implementación de herramienta de cobertura de pruebas unitarias. • Definición del umbral de cobertura.
Despliegue automático	Construcción	2	Si bien el despliegue está automatizado, no inicia automáticamente luego de que finaliza la compilación del código y la ejecución de pruebas unitarias.	• Automatización e integración del despliegue.

Práctica	Etapa	CTIL	Observaciones	Acciones correctivas
Segmentación de pruebas funcionales	Pruebas funcionales	2	Las pruebas funcionales automatizadas solo se agrupan por prioridad y no están segmentadas según las diferentes funcionalidades.	• Relevamiento de las funcionalidades generales y más críticas del sistema. • Segmentación de pruebas funcionales.
Gestión de pruebas de capacidad	Pruebas no funcionales	2	No se tienen umbrales para medir el rendimiento.	• Definición de pruebas de capacidad. • Establecimiento de umbrales.

6.2.4.2. Fase 2: Implementación del PAC

La implementación y corrección de las dos primeras prácticas sugeridas por el PAC (verificación local y construcción), tomaron 1 semana.

Para el primer conjunto de acciones correctivas, el responsable interno (líder técnico), se encargó de seleccionar las herramientas para medir la cobertura de pruebas unitarias, considerando los lenguajes de programación que se utilizan para el desarrollo del sitio. Por un lado, para la capa lógica, se eligió la herramienta NCover, ya que el mismo utilizaba C# como lenguaje principal. Por otro lado, para la capa de interfaz gráfica de usuario, se seleccionó la herramienta Istanbul, recomendada por uno de los expertos para medir la cobertura de aplicaciones Angular. Con la primera ejecución de ambas herramientas se obtuvo una cobertura de 57% para la capa lógica y de 21% para la capa de GUI. Partiendo de estos resultados, los expertos sugirieron definir umbrales que sean alcanzables e ir aumentándolos con el tiempo. Los umbrales de aceptación para la cobertura de pruebas unitarias establecidos fueron:

- Capa lógica: 60%
- Capa de interfaz gráfica de usuario: 30%

La implementación de las herramientas de cobertura tomó 3 días y en los 2 días posteriores, el despliegue del sitio web fue configurado para iniciar de forma automática justo después que finalizan de forma exitosa la construcción del código de forma y la ejecución de pruebas unitarias. Esto fue posible gracias a que las diferentes actividades que conforman el

despliegue ya estaban automatizadas y el servidor de Integración Continua que utilizaba el proyecto (Bamboo) permitía la incorporación de esta etapa como parte de las tareas del proceso de construcción.

Por otro lado, la implementación del resto de las acciones correctivas llevó más tiempo. En primer lugar, se hizo una estimación del esfuerzo requerido para segmentar las pruebas funcionales y para la definición de umbrales de rendimiento. Luego, se desglosaron estas actividades en subtareas que iban siendo incorporadas en cada iteración del proyecto.

En la primera iteración, cada uno de los 6 equipos realizó un relevamiento de las funciones principales de la plataforma. En la segunda, cada equipo comenzó a segmentar los scripts de pruebas en base al relevamiento. Esta tarea continuó durante la tercera iteración, concluyendo con la creación de archivos de lotes de pruebas por cada grupo de pruebas, resultado de la segmentación. En la cuarta iteración, se hizo un análisis de los últimos 3 meses para obtener el tiempo de respuesta promedio de la aplicación, con una carga de 2000 usuarios, durante 30 segundos. Este resultado fue utilizado como el primer umbral de aceptación.

Sin embargo, en la siguiente iteración algunos equipos notaron que las pruebas de rendimiento generaban falsos positivos. Esto condujo a un análisis exhaustivo por parte de los auditores externos y el responsable interno, de las causas por las que se producían los fallos en las pruebas. El resultado de la investigación demostró que factores externos como problemas en uno de los servidores del ambiente, fallas en la conexión y la ejecución de otros programas y servicios en paralelo con las pruebas de rendimiento, hacían que no se cumplan los umbrales establecidos.

Finalmente, los expertos y el líder técnico, en conjunto con el doctorando, decidieron construir un ambiente de pruebas exclusivo para las pruebas de rendimiento. Además, el ambiente se construyó con las mismas especificaciones que el entorno de producción, disminuyendo proporcionalmente los recursos. Esto generó dos ventajas:

- Los falsos positivos dejaron de ocurrir.
- Los resultados eran muy precisos al permitir compararse proporcionalmente a producción.

6.2.4.3. Fase 3: Recolección de datos

Una vez que las acciones correctivas fueron llevadas a cabo satisfactoriamente, se inició el proceso de entrevistas con los auditores externos y el responsable interno. Los resultados de las mismas se presentan a continuación.

- **Pregunta 1*:*** ¿Aplican los tipos de pruebas del proyecto a las etapas de V&V del modelo? Todos los participantes respondieron satisfactoriamente.
- **Pregunta 2*:*** ¿Son interpretadas correctamente cada práctica del modelo para la etapa de V&V y el nivel CTIL en evaluación? Tanto los auditores externos como el líder técnico detectaron una inconsistencia durante la evaluación con la herramienta:
    - En la etapa de pruebas funcionales, la evaluación sugiere que las pruebas funcionales automatizadas se agrupen siguiendo un modelo de capas compuesto por la capa del controlador de aplicación, la capa de pruebas y la capa de criterios de aceptación. Sin embargo, la capa de criterios de aceptación no necesariamente debe ser implementada utilizando una herramienta de BDD. También se puede generar documentación no técnica mediante el uso de logs en cada instrucción de los scripts de pruebas que representen una acción o el cumplimiento de una condición.
- **Pregunta 3*:*** ¿Faltan pasos y/o tareas adicionales para la implementación de cada práctica? Se identificaron las siguientes sugerencias:
    - Uno de los auditores externos señaló que además de segmentar las pruebas funcionales por funcionalidad y prioridad, también se lo puede hacer en base la duración de las pruebas. Por ejemplo, las pruebas más rápidas pueden ejecutarse al inicio y las pruebas más lentas al final. De este modo, se obtiene retroalimentación más rápido.
    - Uno de los expertos y el responsable interno, indicaron que una potencial mejora a la práctica de gestión de pruebas de capacidad sea la implementación del ambiente de pruebas de capacidad proporcional a producción.
- **Pregunta 4:** ¿Hay pasos y/o tareas redundantes o innecesarios en la implementación de cada práctica? Para esta pregunta los participantes no aportaron sugerencias.
- **Pregunta 5:** ¿La implementación de cada práctica asegura la solución parcial de un problema de Pruebas Continuas? Nuevamente, uno de los expertos remarcó que, en el caso de las pruebas de capacidad no solo es necesario la definición del umbral sino también la creación del entorno especializado. Sin ello, no es posible asegurar que los problemas de Pruebas Continuas respecto a las pruebas de rendimiento sean solucionados.

6.2.4.4. Fase 4: Análisis de los resultados y mejora del modelo

La Tabla 67 muestra las mejoras realizadas al modelo para el nivel CTIL abordado en este estudio de caso, las cuales ya se encuentran incluidas en la sección 4.

Tabla 67. Mejoras implementadas para el nivel CTIL 2 luego del tercer ciclo de IA.

Etapa de V&V	Mejora
Pruebas funcionales	En la práctica de **Implementación de un modelo de capas para el marco de trabajo de pruebas funcionales**, se modificó la tarea de **Implementación de la capa de criterios de aceptación** para contemplar no solo herramientas de BDD, sino también cualquier otro tipo de mecanismo, librería o práctica que genere documentación no técnica.
Pruebas no funcionales	Se añade la tarea **Implementación del entorno de pruebas** a la práctica de **Gestión de pruebas de capacidad**.

Asimismo, la recomendación de añadir a la práctica de segmentación de pruebas funcionales el criterio basado en duración de la ejecución no fue incluida. La razón es que la misma ya se encuentra contemplada en un nivel superior.

6.2.5. Ciclo 4: Estudio de caso para el nivel CTIL 3

El tercer estudio de caso se realizó dentro de uno de los proyectos de desarrollo de software de una de las organizaciones de mercados financieros más grande del mundo. La misma opera una bolsa de opciones y futuros. El proyecto seleccionado desarrolla y mantiene cuatro tipos de plataformas:

1. Un sitio web destinado para los usuarios finales que operan en el mercado de futuros y derivados, utilizando como principales tecnologías Java, JSP, nodeJS y React.
2. Un entorno de desarrollo para que editores y autores sin conocimiento de programación puedan publicar contenido en el sitio web. Este entorno se construye utilizando el CMS Adobe Experience Manager (AEM).
3. Dos aplicaciones para dispositivos móviles, una para Android y otra para IOs.
4. Una aplicación para el reloj inteligente de Apple (Apple Watch).

Para cumplir con estos objetivos, el proyecto consta de 3 equipos ágiles. Uno de estos equipos desarrolla el sitio web utilizando la metodología Scrum y está compuesto de 8 desarrolladores, 2 especialistas en pruebas manuales y 2 especialistas en pruebas automatizadas. El segundo equipo utiliza Kanban y se encarga del entorno de desarrollo con AEM. Este se compone de 3 desarrolladores y un especialista en pruebas que ejecuta pruebas manuales y automatizadas. Por último, el tercer equipo desarrolla las aplicaciones para los dispositivos móviles y para el Apple Watch y está compuesto de 2 desarrolladores y 1 especialista en pruebas manuales. También utilizan Kanban entre todos codifican y ejecutan las pruebas automatizadas.

Los miembros de estos equipos son gestionados por un jefe de proyecto, un líder de desarrollo y un líder de control de calidad. El representante del proyecto como responsable interno fue el líder de control de calidad.

6.2.5.1. Fase 1: Evaluación del proceso de pruebas

Antes de comenzar con la evaluación, los auditores externos y el doctorando brindaron una lista de todas las evidencias que se deberían recolectar dentro de los tres equipos del proyecto. Durante una semana, se realizó una búsqueda exhaustiva de evidencias y se decidió comenzar con la evaluación. La Tabla 68 muestra las evidencias requeridas por los auditores y las recolectadas por el proyecto.

Tabla 68. Lista de evidencias requeridas.

Evidencia Requerida	Tipo	Estado
Repositorio donde se encuentra el código fuente	Enlace	Recolectado
Repositorio donde se encuentra el código de las pruebas unitarias	Enlace	Recolectado
Repositorio donde se encuentra el código de las pruebas funcionales	Enlace	Recolectado
Tareas del servidor de IC que se utilizan en el proyecto.	Enlace	Recolectado
Reportes de ejecución de pruebas (conteniendo errores).	Artefacto	Recolectado
Reportes de ejecución de pruebas (exitoso)	Artefacto	Recolectado
Reportes de ejecución de pruebas (con pruebas omitidas)	Artefacto	No encontrado
Reporte de cobertura de pruebas unitarias	Artefacto	Recolectado
Scripts que se ejecutan en el entorno local del desarrollado.	Archivo	No encontrado
Herramienta de análisis estático del código fuente	Enlace	Recolectado

Evidencia Requerida	Tipo	Estado
Reporte de los análisis estáticos del código fuente	Artefacto	Recolectado
Lista de casos de pruebas automatizados (exportados de la herramienta)	Archivo	Recolectado
Acceso a los calendarios del proyecto	Enlace	Recolectado
Minutas de reuniones	Archivo	Recolectado

Con las evidencias recolectadas, los auditores comenzaron la evaluación a través de la herramienta. La evaluación del nivel CTIL 1 se realizó en 1 día y el resultado fue **satisfactorio**. Por otro lado, si bien la evaluación del nivel CTIL 2 llevó aproximadamente 2 semanas, debido al análisis de evidencias, el mismo concluyó también **satisfactoriamente**. Por último, al realizar la evaluación apuntando al nivel CTIL 3, comenzaron a notar falta de evidencias para algunas prácticas y luego de otras 2 semanas de evaluación, el resultado fue **no satisfactorio**. El reporte final se muestra en la Fig. 78.

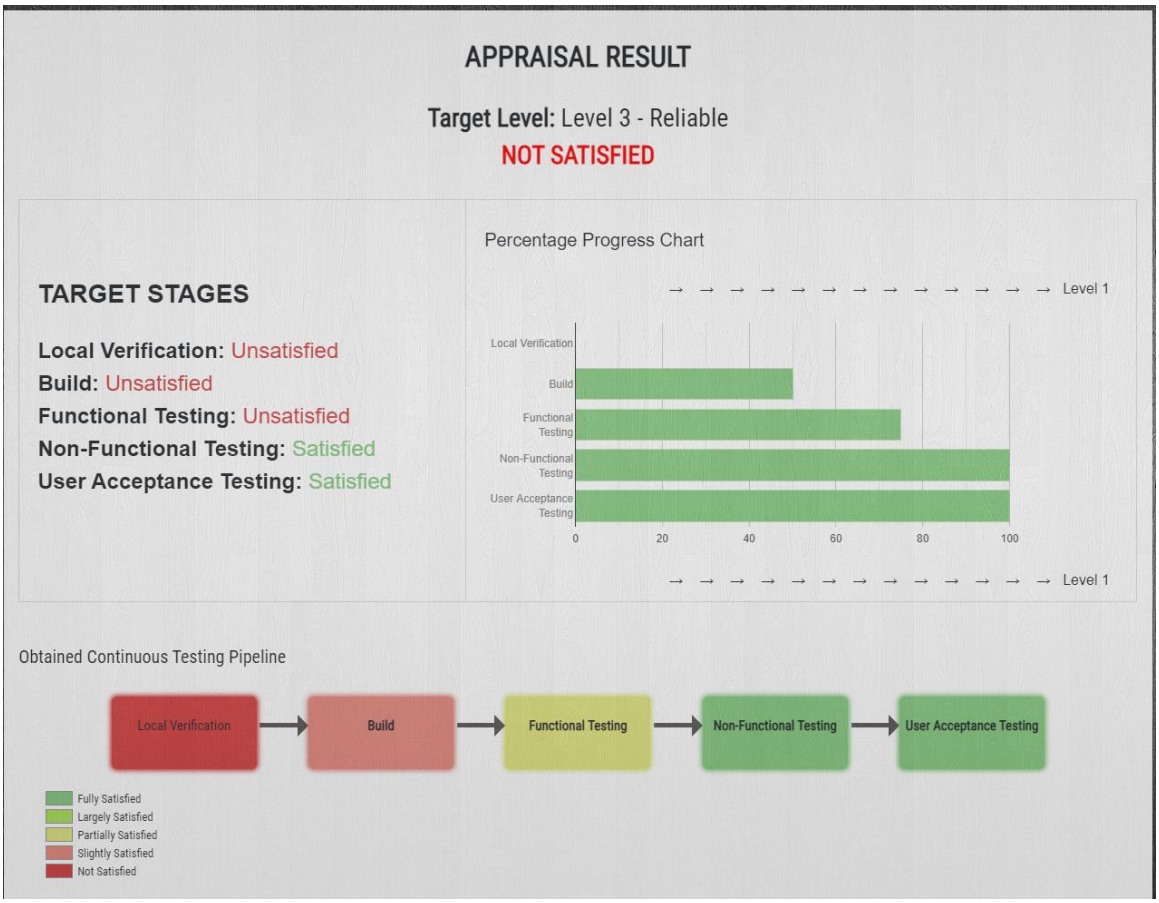


Fig. 78. Resultado de la evaluación final del estudio de caso 3.

El PAC generado por EvalCTIM se muestra en la Tabla 69.

Tabla 69. Acciones correctivas para el estudio de caso 3.

Práctica	Etapa	CTIL	Observaciones	Acciones correctivas
Pruebas simuladas	Verificación local	3	No hay pruebas unitarias que interactúan con los sistemas y/o servicios de terceros.	• Instalación de una herramienta de simulación de pruebas. • Identificación de código sin cobertura que interactúa con componentes externos. • Implementación de pruebas simuladas.
Ejecución de pruebas en memoria.	Verificación local	3	Solo se verifica el resultado de las pruebas funcionales en el servidor de Integración Continua.	• Ejecución de pruebas funcionales antes de integrar el código a la rama principal. • Configuración de pruebas funcionales en memoria.
Pruebas de despliegue e instalación.	Construcción	3	No hay pruebas que verifican que el despliegue haya sido exitoso.	• Creación de pruebas de despliegue e instalación.
Automatización de pruebas especiales	Pruebas funcionales	3	No se automatizaban los escenarios negativos, solo los alternativos.	• Automatización de escenarios negativos.
Estrategia de omisión de pruebas	Pruebas funcionales	3	El proyecto no cuenta con una estrategia de omisión de pruebas.	• Identificación de las causas comunes de errores en la ejecución de los pasos, por falta de datos de pruebas y no cumplimiento de precondiciones. • Definición de la estrategia de omisión de pruebas.

6.2.5.2. Fase 2: Implementación del PAC

A diferencia del resto de los estudios de caso, este proyecto comenzó con la implementación de las acciones correctivas correspondientes a la etapa de pruebas funcionales.

En primer lugar, se actualizó la Definición de Hecho del proyecto para automatizar las pruebas negativas por cada requerimiento. También fue necesario comenzar a incorporar la estimación del esfuerzo requerido para automatizar estas pruebas, durante las etapas de refinamiento y panificación. Uno de los expertos además sugirió aprovechar la oportunidad que se presentó al modificar la Definición de Hecho, para añadir la automatización de las pruebas que se documentaban cuando un defecto surgía. De este modo, a partir de la próxima iteración se comenzaron a automatizar las pruebas negativas y las que se documentaban cuando se detectaban defectos.

Luego, se continuó con la implementación de la estrategia de omisión de pruebas. Una vez identificadas las causas más frecuentes de fallos no relacionados a la verificación principal de cada prueba, el líder de control de calidad elaboró un documento conteniendo el mecanismo de verificación por cada tipo de problemática detectada. Este documento fue utilizado por los diferentes equipos durante un mes para la refactorización de los scripts de pruebas, añadiendo el mecanismo de omisión cuando las precondiciones no se cumplían.

Posteriormente, se continuó con la etapa de verificación local. Antes de comenzar con la implementación de las acciones correctivas para esta etapa, los expertos organizaron una capacitación interna para los 3 equipos de desarrollo sobre el uso de pruebas simuladas. Como el lenguaje de programación principal era Java, el mismo se llevó a cabo utilizando la herramienta Mockito como ejemplo. Luego, en la iteración posterior, se comenzaron a incluir requerimientos de deuda técnica para implementar las pruebas simuladas.

Por último, el líder de control de calidad añadió en la configuración del marco de trabajo de pruebas automatizadas, la opción de ejecutar las pruebas en memoria, utilizando la API HtmlUnitDriver de Selenium WebDriver. Este procedimiento llevó 2 días y después de su implementación, el líder técnico hizo una demostración a los desarrolladores de cada equipo sobre el procedimiento de ejecución de pruebas en memoria en sus ambientes locales. A partir de ese momento, se empezaron a hacer ejecuciones de pruebas funcionales antes de la integración del código al repositorio de control de versiones.

6.2.5.3. Fase 3: Recolección de datos

Para la validación de este nivel CTIL del modelo, al concluir la implementación del PAC se invitó a los auditores externos y al responsable interno, a participar de las entrevistas mencionadas en la sección 6.2.2. Los resultados de las entrevistas se presentan a continuación.

- **Pregunta 1*:*** ¿Aplican los tipos de pruebas del proyecto a las etapas de V&V del modelo? Todos los participantes respondieron que sí.
- **Pregunta 2*:*** ¿Son interpretadas correctamente cada práctica del modelo para la etapa de V&V y el nivel CTIL en evaluación? Mientras que uno de los expertos y el responsable técnico respondieron que las prácticas se interpretan correctamente, el otro experto sugirió lo siguiente:
    - En la etapa de pruebas funcionales, la herramienta de evaluación busca validar si el proyecto se encuentra realizando pruebas de extremo a extremo. Sin embargo, existen dos tipos de pruebas extremo a extremo: las pruebas horizontales y las pruebas verticales. En la descripción de la práctica, parece ser que se hace referencia solamente a las pruebas extremo a extremo horizontales. Se recomienda indicar esta diferenciación en la práctica mencionada, para abarcar ambos tipos.
- **Pregunta 3*:*** ¿Faltan pasos y/o tareas adicionales para la implementación de cada práctica? Se propuso las siguientes modificaciones:
    - Uno de los expertos y el responsable interno, indicaron que el proyecto en evaluación utiliza la tecnología Adobe Experience Manager que también se despliega junto con las otras plataformas. Para verificar que esta tecnología fue despliega con éxito, es necesario validar los llamados “AEM Bundles”. Se sugiere agregar esta tarea a la práctica de pruebas de despliegue e instalación.
    - Uno de los expertos mencionó que fue importante para aumentar la cobertura de pruebas funcionales en la etapa de regresión, la automatización de pruebas que surgieron a partir de defectos. Esto puede ser añadido como un paso adicional en la práctica de automatización de pruebas especiales.
    - Tanto los expertos como el responsable interno señalaron que en la práctica de estrategia de omisión de pruebas no se menciona la refactorización de las pruebas que ya existen. Esta es una actividad clave que debe ser incorporada a la práctica.

- **Pregunta 4:** ¿Hay pasos y/o tareas redundantes o innecesarios en la implementación de cada práctica? Los encuestados no recomendaron mejoras por hacer para esta pregunta.
- **Pregunta 5:** ¿La implementación de cada práctica asegura la solución parcial de un problema de Pruebas Continuas? De forma unánime, los participantes respondieron que las prácticas de este nivel cumplen el objetivo del mismo. Uno de los expertos expresó que "si el objetivo de este nivel es aumentar la confiabilidad de las pruebas, entonces las prácticas de este nivel son adecuadas". Por otro lado, el otro experto añadió que "las prácticas de este nivel representan una recopilación de las mejores propuestas que existen en la literatura para hacer frente a las pruebas no deterministas". Finalmente, el responsable interno expresó que "después de implementar las prácticas sugeridas por el modelo, se eliminaron completamente los falsos positivos en la ejecución de las pruebas".

6.2.5.4. Fase 4: Análisis de los resultados y mejora del modelo

A partir de las respuestas obtenidas en las entrevistas realizadas y al igual que en los estudios de caso anteriores, se llevó a cabo un análisis que concluyó con la inclusión de mejoras en el modelo CTIM. En la Tabla 70, se muestran las mejoras realizadas al modelo en este estudio de caso, como parte del segundo ciclo de IA.

Tabla 70. Mejoras implementadas para el nivel CTIL 3 luego del cuarto ciclo de IA.

Etapa de V&V	Mejora
Construcción	Se modifica el alcance de la práctica **pruebas de despliegue e instalación** a la verificación del estado de cualquier tipo de servidor, servicio, o herramienta, que sea necesaria antes de la ejecución de pruebas.
Pruebas funcionales	En la práctica de **Automatización de pruebas de extremo a extremo**, se dividió la tarea de automatización de escenarios extremo a extremo a: • Automatización de pruebas extremo a extremo horizontales. • Automatización de pruebas extremo a extremo verticales.
Pruebas funcionales	Se añade la tarea **Automatización de pruebas de defectos** a la práctica de **Automatización de pruebas especiales**.
Pruebas funcionales	Se añade la tarea **Refactorización de las pruebas automatizadas con la estrategia de omisión de pruebas** a la práctica **Estrategia de omisión de pruebas**

Sin embargo, la recomendación de añadir específicamente la verificación de “AEM Bundles” a la práctica de pruebas de despliegues e instalación no fue añadida al modelo. Esto se debe a que el objetivo del modelo es brindar prácticas genéricas que contemplen cualquier tipo de proyecto o plataforma que se encuentre dentro del alcance. No obstante, como se muestra en la Tabla 70, el modelo fue modificado para ampliar el alcance de las pruebas de despliegue e instalación para incluir la verificación de cualquier precondición a la ejecución de pruebas.

6.2.6. Ciclo 5: Estudio de caso para el nivel CTIL 4

El último estudio de caso presentado como parte de los ciclos de IA, se llevó a cabo en el área de tecnología de una de las aerolíneas más grandes del mundo y la más grande para vuelos de cabotaje dentro de los Estados Unidos. La evaluación fue realizada dentro del proyecto encargado del desarrollo del sitio web, donde la misma comercializa principalmente la venta de pasajes aéreos.

En el año 2010, este proyecto comenzó la transición a las metodologías ágiles y en el año 2013 comenzó la implementación de prácticas del desarrollo continuo, apoyado por uno de los pioneros de estas prácticas, quién brindaba consultoría. Además, habían certificado CMMI nivel 3, en noviembre 2018.

El proyecto se compone de 2 equipos ágiles. Un equipo realiza el desarrollo de nuevas funcionales para el sitio web y está formado por 8 desarrolladores, 2 especialista en pruebas manuales y 1 analista de negocio. El otro equipo realiza mantenimiento del sitio actual y está formado por 5 desarrolladores y 1 especialista en pruebas. Ambos equipos se encuentran bajo la supervisión de un líder técnico y un jefe de proyecto.

El jefe de proyecto decidió que los equipos participen de la evaluación como parte de la validación de su madurez en Pruebas Continuas y para colaborar con la mejora del modelo propuesto. Finalmente, el líder técnico participó como responsable interno para el estudio de caso.

6.2.6.1. Fase 1: Evaluación del proceso de pruebas

Se procede a realizar la evaluación del modelo CTIM utilizando la herramienta. De forma similar a los otros estudios de caso, los auditores externos participantes solicitaron a los equipos de desarrollo que provean de toda la evidencia posible.

Una vez recopilada toda la evidencia, se realizan las evaluaciones de los diferentes niveles CTIL en forma secuencial, obteniendo los resultados que se muestran en la Tabla 71.

Tabla 71. Resultados de las evaluaciones realizadas en el estudio de caso 4.

CTIL	Resultado
1	Satisfactorio
2	Satisfactorio
3	Satisfactorio
4	No Satisfactorio

Los detalles de la evaluación final se muestran en la Fig. 79.

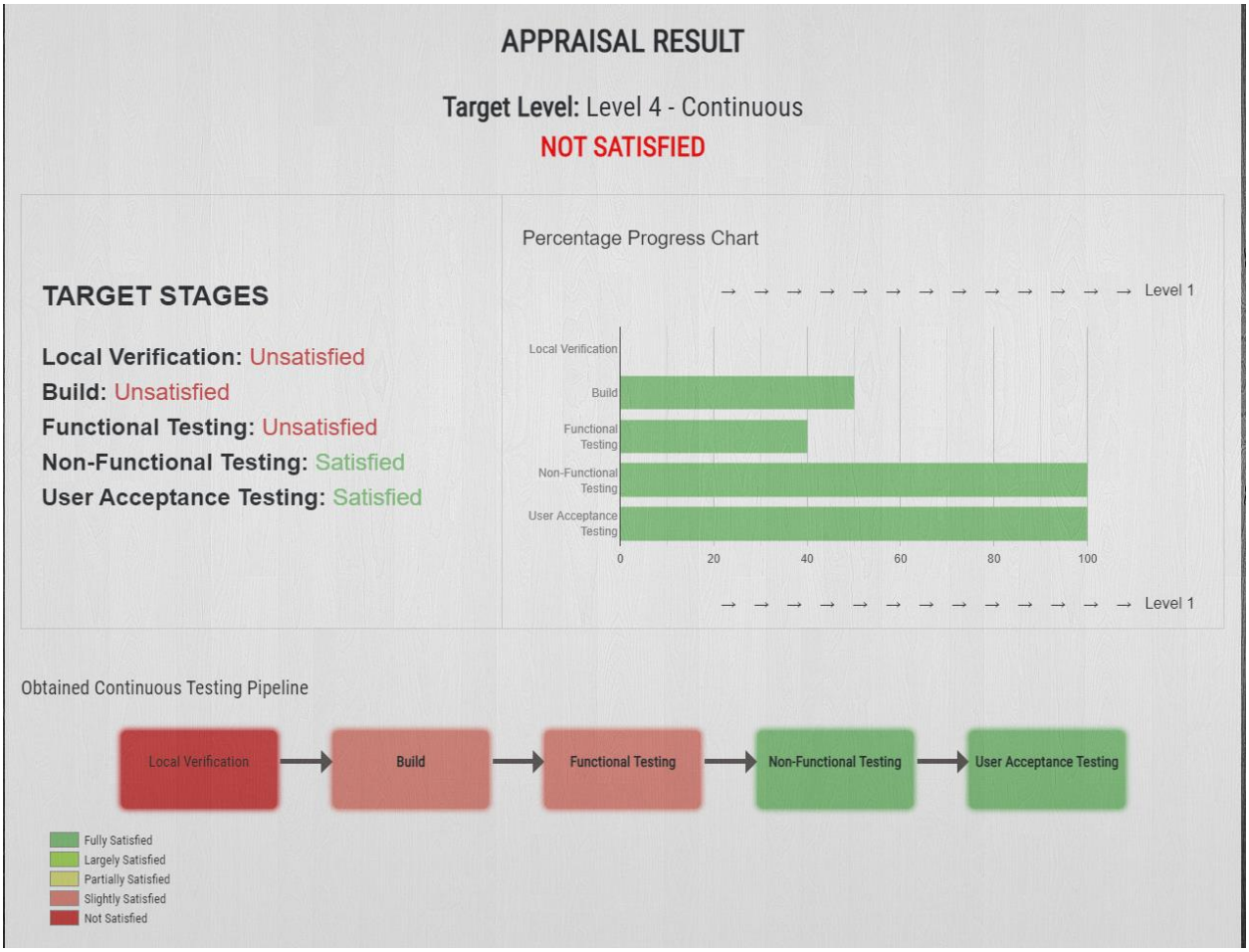


Fig. 79. Resultado de la evaluación final del estudio de caso 4.

La Tabla 72 muestra las acciones correctivas para el estudio de caso 4.

Tabla 72. Acciones correctivas para el estudio de caso 4.

Práctica	Etapa	CTIL	Observaciones	Acciones correctivas
Pruebas unitarias en segundo plano	Verificación Local	4	Las pruebas unitarias no se ejecutan en segundo plano.	• Creación del script de ejecución de pruebas unitarias en segundo plano.
Escalabilidad y rendimiento de la construcción	Construcción	4	No se toman métricas para mejorar la etapa de construcción.	• Definición y obtención de métricas. • Análisis de métricas de la etapa de construcción y ejecución de un plan de acción.
Selección de pruebas a ejecutar	Pruebas funcionales	4	La totalidad de las pruebas se ejecutan por cada cambio introducido.	• Lectura de mensajes de confirmación de cambios al repositorio de control de versiones. • Ejecución de pruebas en base a los mensajes de las confirmaciones de cambios.
Comparación de imágenes	Pruebas funcionales	4	Sin observaciones	• Implementar un algoritmo de comparación de imágenes.
Monitorización continua de pruebas funcionales	Pruebas funcionales	4	No se toman métricas para mejorar la etapa de pruebas funcionales.	• Definición y obtención de métricas. • Análisis de métricas de la etapa de pruebas funcionales y ejecución de un plan de acción.

6.2.6.1. Fase 2: Implementación del PAC

En primer lugar, los expertos decidieron comenzar por las acciones correctivas correspondientes a las prácticas de escalabilidad y rendimiento de la construcción y

monitorización continua de pruebas funcionales. Para ello, definieron métricas útiles para el proyecto partiendo de las recomendadas por Duvall, Matyas y Glover [9]. Las métricas seleccionadas se muestran en la Tabla 73.

Tabla 73. Métricas para la monitorización continua de la construcción y pruebas funcionales del estudio de caso 4.

Métrica	Etapa
Tiempo total de la etapa de construcción	Construcción
Tiempo de ejecución de las pruebas	Pruebas funcionales
Tiempo de ejecución de las pruebas unitarias	Construcción
Tiempo de la ejecución de inspecciones	Construcción
Razón entre construcciones exitosas y fallidas.	Construcción
Razón entre ejecuciones exitosas y fallidas.	Pruebas funcionales
Tiempo total de despliegue.	Construcción
Uso de recursos del servidor de IC en la construcción	Construcción
Uso de recursos del servidor de IC en la ejecución de pruebas funcionales.	Pruebas funcionales
Cantidad de pruebas no deterministas por ejecución.	Pruebas funcionales

Una vez definidas las métricas, el líder técnico destinó dos semanas a realizar pruebas de concepto, instalando diferentes plugins para el servidor de Integración Continua, que permite monitorizar las diferentes métricas. Luego, destinó otras dos semanas para integrar los resultados de estos plugins con una herramienta de visualización de datos llamada Kibana. Esta herramienta muestra en tiempo real la monitorización de las diferentes métricas.

Posteriormente, se continuó con la creación del script para la ejecución de pruebas unitarias en segundo plano. Para ello, los expertos recomendaron desarrollar un script de bash para invocar a las pruebas unitarias de forma programada. El líder técnico realizó la codificación de este script en 1 día y luego lo programó para que se ejecute de forma continua, tomando con cada ejecución los cambios nuevos introducidos por el desarrollador. Luego, el script se distribuyó a ambos equipos y lo comenzaron a implementar inmediatamente.

Para las acciones correctivas de la etapa de pruebas funcionales, primero avanzaron con la ejecución de pruebas segmentadas. Como las pruebas ya se encontraban agrupadas por funcionalidad, el desafío era generar un mecanismo que permita identificar la funcionalidad que se está modificando con los cambios a introducirse en el repositorio de control de versiones. En

primer lugar, los expertos propusieron el uso de un estándar para los mensajes en las confirmaciones de cambios, donde se indique la funcionalidad afectada. Luego, el líder técnico desarrolló un script que fue añadido a la fase de pruebas funcionales del servidor de Integración Continua. El script leía el mensaje de la confirmación de cambios y obtenía del mismo la funcionalidad afectada mediante el uso de expresiones regulares. De este modo, fue posible la ejecución de pruebas automatizadas específicas según el mensaje en la confirmación de cambios. El desarrollo de este script llevó 2 semanas más.

Por último, se implementó la práctica de comparación de imágenes, utilizando el algoritmo de comparación de imágenes propuesto en [123]. La adaptación de este algoritmo al proceso de pruebas del proyecto de desarrollo llevó 2 días. Sin embargo, la introducción del código para manipular las secciones más importantes del sitio web que iban a ser verificadas con el algoritmo llevó 4 semanas.

6.2.6.2. Fase 3: Recolección de datos

Los auditores externos y el responsable interno participaron de las entrevistas mencionadas en la sección 6.2.2. Los resultados se presentan a continuación.

- **Pregunta 1*:*** ¿Aplican los tipos de pruebas del proyecto a las etapas de V&V del modelo? Todos los participantes respondieron satisfactoriamente.
- **Pregunta 2:** ¿Son interpretadas correctamente cada práctica del modelo para la etapa de V&V y el nivel CTIL en evaluación? Nuevamente, todos los encuestados respondieron que sí.
- **Pregunta 3*:*** ¿Faltan pasos y/o tareas adicionales para la implementación de cada práctica? En las respuestas a esta pregunta, se identificaron las siguientes recomendaciones:
    - El líder técnico (responsable interno) sugirió para la práctica de selección de pruebas a ejecutar que independientemente del grupo de pruebas que fue seleccionado para ejecutarse, siempre debe ir acompañado de pruebas de humo críticas.
    - Uno de los expertos recomendó para la práctica de pruebas funcionales en paralelo que también se puede considerar el uso de servicios en la nube como BrowserStack o Sauce Labs para ejecutar las pruebas en paralelo.

- Los dos auditores externos recomendaron para la práctica de monitorización continua de pruebas funcionales incluir la métrica de cantidad de pruebas no deterministas por cada ejecución.
- Para la etapa de pruebas no funcionales en la práctica de automatización de pruebas de capacidad, el líder técnico y otro de los expertos expresaron que uno de los aspectos más importantes a considerar al automatizar pruebas de capacidad es si las interacciones con el sistema se realizan mediante la GUI o mediante una servicio o API. Si se está desarrollando un sistema que es o será muchos usuarios, será imposible generar carga al sistema mediante interacciones con la GUI. Esto se debe a que, para simular un escenario real, se necesitaran miles de máquinas clientes interactuando con el sistema. En cambio, utilizando las API, se podría hacerlo. Se sugiere mostrar todos los puntos potenciales para realizar pruebas de capacidad automatizadas.

- **Pregunta 4:** ¿Hay pasos y/o tareas redundantes o innecesarios en la implementación de cada práctica? Tanto los expertos como el responsable interno no consideraron mejoras por hacer.
- **Pregunta 5:** ¿La implementación de cada práctica asegura la solución parcial de un problema de Pruebas Continuas? Uno de los expertos y el líder técnico recomendaron añadir la siguiente práctica:
  - Rotación de navegadores. Rotar los navegadores entre las pruebas en paralelo logra gradualmente la misma cobertura que ejecutar las mismas pruebas en un solo navegador. De este modo, no es necesario repetir la ejecución de todas las pruebas por cada navegador, ya que, variando los navegadores para los escenarios secundarios, se tiene la misma cobertura y se ahorra tiempo de ejecución.

6.2.6.3. Fase 4: Análisis de los resultados y mejora del modelo

Al igual que en los otros estudios de caso, a partir de las respuestas obtenidas en las entrevistas realizadas, se llevó a cabo el análisis correspondiente. Este concluyó con la inclusión de mejoras en el modelo CTIM. Como se mencionó anteriormente, estas ya se encuentran reflejadas en la versión final del modelo presentado en la sección 4, junto con el resto de las mejoras previamente relevadas.

En la Tabla 74, se muestran las mejoras realizadas al modelo en este estudio de caso, como parte del segundo ciclo de IA.

Tabla 74. Mejoras implementadas para el nivel CTIL 4 luego del quinto ciclo de IA.

Etapa de V&V	Mejora
Pruebas funcionales	Se añade la práctica **Rotación de navegadores**
Pruebas funcionales	En la práctica **selección de pruebas a ejecutar** se añade la ejecución de pruebas de humo, independientemente del grupo seleccionado de pruebas a ejecutar.
Pruebas funcionales	Se añade la alternativa de **servicios en la nube** como una alternativa para construir la infraestructura de paralelización de pruebas funcionales, en la práctica **pruebas funcionales en paralelo**
Pruebas funcionales	Se añade la métrica **cantidad de pruebas no deterministas por ejecución** a la tabla de métricas para la práctica de **monitorización continua de pruebas funcionales**.
Pruebas no funcionales	Se añade en la práctica de **automatización de pruebas de capacidad** los puntos potenciales de inyección para pruebas de capacidad.

6.3. Total de mejoras realizadas al modelo

En la Tabla 75 se presenta un resumen de todas las mejoras realizadas al modelo.

Tabla 75. Total de mejoras realizadas al modelo.

CTIL	Mejoras realizadas	Propuestas no realizadas	Prácticas añadidas	Prácticas modificadas	Prácticas eliminadas
1	6	3	0	4	0
2	6	4	1	3	0
3	11	4	4	5	0
4	7	3	1	4	0

En todos los niveles se realizaron mejoras y modificaron las buenas prácticas. El nivel CTIL que tuvo mayor cantidad de mejoras realizadas fue el CTIL 3, donde se añadieron 4 nuevas prácticas sugeridas por los expertos durante el primer ciclo de IA (sección 6.1.4). También se añadieron una práctica en el nivel CTIL 2 y otra en el nivel CTIL 4. Finalmente, en ninguno de los niveles CTIL se eliminaron prácticas de Pruebas Continuas.

7. CONCLUSIÓN

Como conclusión de esta Tesis Doctoral, en este capítulo se analiza el cumplimiento de los objetivos establecidos en la sección 1.2 y se resumen las principales contribuciones realizadas.

Al inicio se presentaron una serie de objetivos específicos (sección 1.2.1) para cumplir con el objetivo principal: la construcción de un modelo, compuesto por distintas etapas y niveles de mejora, que permita la implementación de Pruebas Continuas y apoyen el desarrollo continuo de software.

En primer lugar, **O1** buscaba identificar y relevar los problemas existentes relacionados a las pruebas de software en los procesos de desarrollo continuo, tanto en la literatura académica como en la industria. Para ello, en primer lugar se llevó a cabo una revisión sistemática de literatura (sección 3.2) donde se obtuvo una lista de 8 problemas sin solución para Pruebas Continuas: pruebas de aplicaciones basadas en servicios en la nube, desafíos con pruebas de GUI, monitoreo continuo, TaaS en Entrega Continua, pruebas automatizadas de microservicios, pruebas no deterministas, pruebas de Big Data y las pruebas de contenido web dinámico. En segundo lugar, se utilizó una encuesta (sección 3.4) para recabar información acerca de los problemas existentes relacionados a Pruebas Continuas en la industria. Los resultados de la encuesta relevaron los siguientes problemas:

- Pruebas no deterministas.
- Automatización de pruebas de interfaz gráfica de usuario.
- Automatización de pruebas web con contenido dinámico.
- Resultados de ejecución de pruebas ambiguos.
- Pruebas en Big Data.
- Pruebas de datos.
- Pruebas en dispositivos móviles.
- Pruebas de servicios web.
- Pruebas que consumen mucho tiempo de ejecución.
- Falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo.
- Ambientes inestables.
- Pruebas automatizadas de requerimientos no funcionales.
- Pruebas de aplicaciones compuestas por servicios en la nube.

Luego, **O2** se definió para analizar los últimos avances, propuestas, herramientas, enfoques y soluciones existentes para los problemas detectados. La revisión sistemática de la literatura descrita en la sección 3.2 también tenía como cuestión de investigación la identificación de soluciones parciales, herramientas y propuestas para hacer frente a los problemas de Pruebas Continuas. Asimismo, durante el desarrollo de esta Tesis Doctoral se elaboraron propuestas para diferentes problemáticas relacionadas con el proceso de pruebas en entornos de Despliegue Continuo y Entornos Continuo. Entre las propuestas desarrolladas se encuentran:

- Herramientas para la evaluación de la usabilidad [100]–[102].
- Un enfoque para la detección de incompatibilidades web [8], [123]–[125].
- Un patrón de diseño para la creación y ejecución de pruebas automatizadas en servicios REST y microservicios [154], [155].
- Un mecanismo para disminuir las pruebas no deterministas [20].

Tanto las propuestas encontradas con la revisión sistemática de la literatura como las desarrolladas, fueron incluidas en el modelo propuesto. Esto forma parte de **O3**: construir un modelo de Pruebas Continuas, mediante la combinación de las técnicas y herramientas existentes, como así también de las diferentes propuestas que existen en la investigación académica. El modelo propone cuatro niveles de mejoras y contempla un conjunto de etapas de verificación y validación (V&V) que a su vez abarcan las propuestas, herramientas y técnicas reportadas como un conjunto de buenas prácticas.

El nivel de mejora 1 del modelo se llama implementación. Propone la automatización de pruebas como el primer paso en la adopción de Pruebas Continuas y, la implementación inicial de cada una de las etapas del Conducto de Pruebas Continuas.

El nivel de mejora 2 se denomina gestión. En el mismo se implementan buenas prácticas relacionadas principalmente con la gestión de las pruebas. Las pruebas unitarias y las funcionales son agrupadas. Se mide la cobertura de las pruebas unitarias, y el marco de trabajo de pruebas funcionales se divide en capas para facilitar su mantenibilidad. También se propone la utilización de una granja para las pruebas en dispositivos móviles y la adopción de pruebas exploratorias y de capacidad. Este nivel soluciona en gran medida los problemas de baja cobertura de pruebas en Entregas Continua.

El nivel de mejora 3 del modelo se llama confiabilidad. El problema que se resuelve principalmente en este nivel es el de las pruebas no deterministas. Se definen estándares y patrones para un proceso de pruebas que genere resultados confiables. Entre las buenas prácticas se encuentran la implementación de pruebas simuladas, la ejecución de pruebas

funcionales en memoria y de forma programada, el análisis estático del código fuente, la ejecución de pruebas de despliegue, la gestión de los tiempos de espera y de los datos, la automatización de tipos especiales de pruebas, estrategias de omisión de pruebas y la mejora de reportes.

El nivel de mejora 4 se denomina continuidad. Se implementan prácticas como selección de pruebas, paralelización, uso de las API para ejecutar precondiciones, rotación de navegadores, entre otras. En este nivel se resuelve el problema de los tiempos de ejecución elevados. También busca incorporar mecanismos de mejora continua para los diferentes tipos de pruebas.

Posteriormente, **O4** buscaba desarrollar una herramienta para evaluar el proceso de pruebas de una organización en base al modelo construido. La herramienta EvalCTIM fue desarrollada de forma iterativa-incremental, como una plataforma web que soporta la evaluación del proceso de Pruebas Continuas utilizando el modelo propuesto (sección 5.2). Además, la misma genera reportes con resultados detallados acerca del cumplimiento de cada buena práctica y elabora un Plan de Acciones Correctivas con recomendaciones para subsanar las no conformidades y aspectos por mejorar hallados durante el proceso de evaluación.

Por último, **O5** buscaba implementar y validar el modelo en ambientes reales de desarrollo, que utilicen o busquen el enfoque de desarrollo continuo de software. Para ello, se utilizó el método de Investigación-Acción.

Antes de implementar el modelo se hizo una validación teórica del mismo (sección 6.1), donde un grupo de participantes expertos propusieron mejoras. Luego de esta primera etapa, el modelo se implementó en diez proyectos pertenecientes a empresas que desarrollaban software, de donde cuatro de ellos fueron descritos como estudios de caso en la sección 6.2 y cuyos resultados permitieron realizar 30 mejoras. Dentro de estas mejoras, se añadieron 4 nuevas buenas prácticas al nivel 3 del modelo, 1 en el nivel 1 y 1 más en el nivel 4. También se modificaron 16 buenas prácticas de acuerdo a las recomendaciones de los expertos. Por último, no se eliminó ninguna práctica desde la versión inicial del modelo.

Tanto los expertos que fueron seleccionados para la primera etapa de validación como especialistas que participaron de la implementación del modelo en empresas reales, expresaron su conformidad con el modelo como una solución a los problemas de Pruebas Continuas que existen en la actualidad. Sin embargo, existen una serie de tareas determinadas como trabajos futuros, las cuales se listan a continuación:

- Continuar identificando nuevos problemas en Pruebas Continuas: la evolución de los mercados y la competencia hace que las empresas que desarrollan software en entornos continuos tengan que incorporar nuevos procedimientos y herramientas en sus procesos de desarrollo y eso genera la aparición de nuevos problemas relacionados con la calidad del software.
- Ampliar el modelo de mejora para Pruebas Continuas: actualmente se están realizando pruebas piloto con nuevos enfoques para acelerar la ejecución de pruebas funcionales y mejorar la gestión de las pruebas no deterministas.
- Investigar y desarrollar soluciones para las pruebas en Big Data: uno de los problemas existentes es la falta de propuestas y herramientas para realizar pruebas en plataformas que utilicen Big Data.
- Mejorar la evaluación y la herramienta EvalCTIM: actualmente se están realizando mejoras en el proceso de evaluación y la herramienta EvalCTIM, incluyendo guías de mejoras y funciones adicionales como, por ejemplo, la comparación de diferentes evaluaciones realizadas. Estas mejoras serán sometidas a nuevas etapas de experimentación.

ANEXO A: Revisión Sistemática de la Literatura

En este anexo se detallan las fases previas al análisis de resultados de la RSL presentado en la sección 3.2. También se listan los artículos seleccionados y la evaluación de la calidad de los mismos.

Planificación y estrategia de búsqueda

Una vez establecidas las cuestiones de investigación que se mencionaron en la sección 3.2, el siguiente paso en una RSL es definir los criterios de búsqueda. Las etapas fueron:

- Realización de una búsqueda preliminar para encontrar una revisión sistemática similar a la que va a ser realizada y conocer el volumen de resultados que pueden obtenerse relacionados sobre el tema de la revisión.
- Realización de diferentes búsquedas de prueba para encontrar términos relacionados que puedan ser incluidos en la búsqueda.
- Identificación de sinónimos y alternativas a los términos de búsqueda encontrados, para aumentar los resultados obtenidos.
- Utilización de operadores booleanos AND y OR entre los distintos términos, para optimizar las búsquedas.

En primer lugar, se realizó una búsqueda preliminar para conocer otras investigaciones relacionadas. A partir de esto, se identificaron sinónimos y alternativas a EC. Según Laukkanen et al. [12], "Entrega Continua es un tema nuevo" y no hay mucha literatura sobre pruebas en entornos continuos, por lo que se decide incluir Integración Continua y Despliegue Continuo para ampliar el alcance de la búsqueda. Esto se debe a que los mismos son prácticas previas y extensiones de Entrega Continua [7]. Por lo tanto, la consulta de búsqueda utilizada fue:

“(((“Continuous Development” OR “Continuous Integration” OR “Continuous Deployment” OR “Continuous Delivery” OR “Rapid Releases”) AND (Testing OR Test)) OR “Continuous Testing”) AND Software”

La primera parte de la consulta busca estudios en el campo de Entrega Continua, sus sinónimos (desarrollo continuo y lanzamientos rápidos) y los otros términos incluidos (IC y DC). La segunda parte, intenta identificar, evaluar y sintetizar toda la literatura disponible relevante para Pruebas Continuas. Finalmente, la palabra "software" al final se incluyó para excluir artículos

que no están relacionados con la ingeniería de software; el mismo enfoque se usó en RSL anteriores [12], [131].

La cadena de búsqueda se aplicó a títulos, resúmenes y palabras clave. La selección de las bases de datos se realizó en base a su utilización en otras RSL anteriores en ingeniería de software [12], [192], [193] y contienen publicaciones que se consideran relevantes para el área de interés. Utilizando la cadena de búsqueda mencionada, se obtuvieron una gran cantidad de resultados, pero muchos de ellos se consideraron irrelevantes para el propósito de este estudio. La Tabla 76 muestra las bases de datos utilizadas como fuentes de datos y el resumen de los resultados obtenidos.

Tabla 76. Bases de datos utilizadas como fuentes de datos con los resultados obtenidos.

Base de datos	Total	Descartes			Segunda búsqueda	Incluidos
		Por repetición	Por resumen	Por irrelevancia		
Scopus	272	243	172	26	3	29
IEEE Xplore	182	96	58	11	2	13
ISI Web of Science	81	33	22	5	1	6
ACM Digital Library	71	35	29	4	1	5
Science Direct	34	19	8	2	0	2
Research at Google	15	13	6	0	1	1
Total	655	439	295	48	8	56

Estrategia de filtrado

La primera búsqueda generó un total de 655 resultados como se ve en la Tabla 76. Esos resultados se filtraron utilizando diferentes criterios de inclusión y exclusión. Para los primeros descartes, se utilizó el siguiente criterio de exclusión:

1) Criterio de exclusión: los estudios duplicados se descartan.

De los 655 artículos, se eliminaron los estudios duplicados, dejando un total de 439 artículos. Luego, se estudiaron los resúmenes de los trabajos restantes y se aplicaron los siguientes criterios de inclusión y exclusión:

2) Criterio de inclusión: se incluyen artículos que proponen herramientas, marcos de trabajo, buenas prácticas o cualquier tipo de solución para una práctica continua de desarrollo de software.
3) Criterio de inclusión: se incluyen artículos que estudian Pruebas Continuas.
4) Criterio de exclusión: si una práctica continua de desarrollo de software o un tema de Pruebas Continuas no se menciona en el resumen, entonces se descarta.

Un total de 295 artículos restaron luego de aplicar los criterios. Posteriormente, se adquirieron las versiones completas de los estudios, aplicando el último criterio de exclusión:

5) Criterio de exclusión: los artículos que no responden a ninguna de las preguntas de investigación se descartan.

Después de aplicar los cinco criterios, se obtuvieron un total de 48 artículos. Como este proceso de filtrado se aplicó durante cinco meses, se repitió el proceso para los artículos que se publicaron durante esos meses. Finalmente, se incluyeron un total de 56 artículos. Con el objetivo de registrar con precisión cualquier información necesaria para responder las preguntas de investigación, se utilizó el formulario descrito en la Tabla 77 con cada uno de los artículos. Este formulario de extracción se ha utilizado previamente en otras RSL de ingeniería de software [194].

Tabla 77. Formulario de extracción.

#	Información del artículo	Descripción	Objetivo
1	Identificador del estudio	Único identificador del estudio	Resumen del estudio
2	Título, autor, año y país de publicación.		Resumen del estudio
3	Base de datos o fuente de extracción.	Scopus, IEEE Xplore, ISI Web of Science, ACM Digital Library, Science Direct, Google Research.	Resumen del estudio
4	Tipo de artículo.	Revista, artículo de conferencia / simposio / taller, capítulo de libro, libro completo, etc.	Resumen del estudio
5	Contexto de aplicación.	Industria, académico, ambos.	Resumen del estudio
6	Tipo de investigación.	Investigación de validación, investigación de evaluación, propuesta o solución, artículo filosófico, trabajo experimental.	Resumen del estudio

#	Información del artículo	Descripción	Objetivo
7	Método de evaluación.	Experimento controlado, caso de estudio, encuesta, etnografía, investigación de acción, revisión sistemática de la literatura, no aplicable.	Resumen del estudio
8	Pruebas Continuas.	¿Existe una definición válida y aceptada para las Pruebas Continuas?	Cuestión 1
9	Niveles de pruebas.	¿Qué tipos de pruebas o niveles de pruebas han sido implementadas en proyectos de desarrollo de software continuo?	Cuestión 2
10	Soluciones para los problemas en la etapa de pruebas.	¿Qué soluciones han sido reportadas para resolver los problemas relacionados a las pruebas en los proyectos de desarrollo continuo?	Cuestión 3
11	Problemas abiertos relacionados a las pruebas en desarrollo continuo.	¿Existen todavía problemas de pruebas en proyectos de desarrollo continuo?	Cuestión 4

Artículos seleccionados

Esta sección describe los resultados de la RSL, donde las respuestas de cada pregunta de investigación son discutidas individualmente.

Como se describió anteriormente, el proceso de selección resultó en 56 estudios que cumplieron con los criterios de inclusión. Además, los datos fueron extraídos utilizando el formulario de extracción descrito en la Tabla 77. A continuación, en la Tabla 78 se enumeran los artículos seleccionados.

Tabla 78. Artículos seleccionados.

#	Ref	Nombre del artículo
S1	[195]	A complete automation of unit testing for JavaScript programs
S2	[196]	A Practical Approach to Software Continuous Delivery Focused on Application Lifecycle Management
S3	[197]	A Scalable Big Data Test Framework

#	Ref	Nombre del artículo
S4	[198]	A survey on quality assurance techniques for Big Data applications
S5	[126]	Ajax best practice: Continuous testing
S6	[199]	An approach for service composition and testing for cloud computing
S7	[147]	An Empirical Analysis of Flaky Tests
S8	[200]	Automated Test Results Processing
S9	[201]	Automated testing in the continuous delivery pipeline: A case study of an online company
S10	[151]	Continuous Architecture and Continuous Delivery
S11	[13]	Continuous Delivery: Huge Benefits, but Challenges Too
S12	[18]	Continuous delivery practices in a large financial organization
S13	[5]	Continuous Deployment of Mobile Software at Facebook
S14	[164]	Continuous Software Testing in a Globally Distributed Project
S15	[202]	Continuous Test Generation on Guava
S16	[203]	Continuous test-driven development: A novel agile software development practice and supporting tool
S17	[204]	Continuous testing in eclipse
S18	[183]	Crowdsourcing GUI tests
S19	[128]	Data debugging with continuous testing
S20	[205]	Designing an Android continuous delivery pipeline
S21	[171]	Determining flaky tests from test failures
S22	[165]	DevOps Advantages for Testing: Increasing Quality through Continuous Delivery
S23	[206]	Effect of time window on the performance of continuous regression testing
S24	[207]	Failure history data-based test case prioritization for effective regression test
S25	[178]	Fast feedback from automated tests executed with the product build
S26	[177]	Hard Problems in Software Testing Solutions Using Testing as a Service (TaaS)
S27	[179]	Improving Web User Interface Test Automation in Continuous Integration
S28	[208]	Industrial application of visual GUI testing: Lessons learned
S29	[209]	Intelligent Testing System
S30	[172]	Large-scale test automation in the cloud
S31	[210]	Learning for Test Prioritization: An Industrial Case Study
S32	[211]	Microservices validation: Mjolnirr platform case study

#	Ref	Nombre del artículo
S33	[212]	Model-Based Continuous Integration Testing of Responsiveness of Web Applications
S34	[213]	Multi-Perspective Regression Test Prioritization for Time-constrained Environments
S35	[214]	Novel Framework for Automation Testing of Mobile Applications using Appium
S36	[215]	O!Snap: Cost-Efficient Testing in the Cloud
S37	[216]	On the long-term use of visual gui testing in industrial practice: A case study
S38	[217]	Perceptual Difference for Safer Continuous Delivery
S39	[218]	Principles of Continuous Architecture
S40	[219]	Prioritizing Manual Test Cases in Traditional and Rapid Release Environments
S41	[12]	Problems, causes and solutions when adopting continuous delivery—A systematic literature review
S42	[220]	Reducing wasted development time via continuous testing
S43	[221]	Regression and Performance Testing of an e-learning Web Application - dotLRN
S44	[222]	Supporting Continuous Integration by Code-Churn Based Test Selection
S45	[223]	Techniques for Improving Regression Testing in Continuous Integration Development Environments
S46	[224]	Test case prioritization for continuous regression testing: An industrial case study
S47	[225]	Test Orchestration: A framework for Continuous Integration and Continuous Deployment
S48	[226]	Test prioritization with optimally balanced configuration coverage
S49	[227]	Test-Driven Development of Relational Databases
S50	[228]	Testing in parallel: A need for practical regression testing
S51	[229]	The State of Continuous Integration Testing @Google
S52	[230]	TITAN: Test Suite Optimization for Highly Configurable Software
S53	[180]	Understanding DevOps & bridging the gap from continuous integration to continuous delivery
S54	[231]	Using acceptance tests to validate accessibility requirements in RIA
S55	[232]	Verdict Machinery: On the Need to Automatically Make Sense of Test Results
S56	[176]	Virtual to the (Near) End: Using Virtual Platforms for Continuous Integration

Antes de presentar los resultados y el análisis de cada cuestión de investigación, se describen los resultados de la evaluación de calidad de los artículos y se brinda una descripción general de las características generales de ellos.

Evaluación de la calidad de los resultados

La evaluación de la calidad es un factor clave para aumentar la confiabilidad y credibilidad de las conclusiones. Dermeval et. al [233], proponen una lista de diez preguntas de evaluación de calidad de artículos, las cuáles brindan una media que representa una manera de medir si los mismos pueden hacer una valiosa contribución a la RSL. Estas preguntas se presentan en la Tabla 79.

Tabla 79. Criterios de evaluación de la calidad de los artículos **[233]**.

Id	Pregunta
Q1	¿Son claros los objetivos de investigación del artículo?
Q2	¿Se describe con claridad la propuesta del artículo?
Q3	¿Existe una descripción adecuada del contexto (industria, entorno de laboratorio, productos utilizados, etc.) en el que se realizó la investigación?
Q4	¿Es la muestra representativa de la población a la que se generalizarán los resultados?
Q5	¿Fue el análisis de datos lo suficientemente riguroso?
Q6	¿Hay una discusión sobre los resultados del estudio?
Q7	¿Se discuten explícitamente las limitaciones de la investigación?
Q8	¿Son interesantes y/o relevantes las lecciones aprendidas para otros profesionales?
Q9	¿Hay suficiente discusión sobre los trabajos relacionados? (Por ejemplo: ¿Se discuten y comparan las técnicas de otros estudios con la técnica actual?)
Q10	¿Qué tan claros son los supuestos, perspectivas teóricas, opiniones o valores que le han dado forma a la investigación?

A continuación, en la

Tabla 80 se presentan los resultados de la evaluación de calidad de los estudios seleccionados, de acuerdo con las preguntas de evaluación descritas en la Tabla 79.

Tabla 80. Evaluación de la calidad de los artículos seleccionados para RSL.

ID	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Puntuación Total	Calidad (%)	Citas
S1	1	1	1	0	0.5	1	0.5	0	0	1	6	60%	11
S2	1	1	1	0	1	0.5	0.5	0	1	1	8	70%	0
S3	1	1	1	0	0	0	0	0	0	1	4	40%	5
S4	1	1	1	1	1	1	1	0.5	0.5	0.5	8.5	85%	0
S5	0	0	1	0	0	0	0	0	0	0.5	1.5	15%	0
S6	1	1	0.5	1	1	0	0	1	1	1	7.5	75%	37
S7	1	1	1	1	1	1	1	1	0	1	9	90%	31
S8	1	1	0	0.5	1	0.5	0	0.5	0	1	5.5	55%	1
S9	1	1	1	1	1	1	1	1	0	1	9	90%	8
S10	1	1	1	0	0	1	1	1	0	1	7	70%	3
S11	1	1	1	0	0	0	1	1	1	1	7	70%	73
S12	1	1	1	0.5	1	1	1	1	1	1	1	95%	4
S13	1	1	1	0	0	0	1	1	1	1	7	70%	1
S14	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	6
S15	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	13
S16	1	1	1	0	1	1	1	1	1	1	9	90%	8
S17	1	1	1	0	0.5	0.5	0.5	1	1	1	7.5	75%	58
S18	1	1	1	0.5	1	0.5	1	1	1	1	8.5	90%	18
S19	1	0.5	1	0	0	0	0	0	1	1	4.5	45%	13
S20	1	0.5	0.5	1	1	1	1	0.5	1	0.5	8	80%	3
S21	1	1	1	0	1	1	1	0.5	1	1	8.5	85%	0
S22	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	2
S23	1	1	1	0	1	1	1	1	1	1	7	90%	0
S24	1	1	1	0	0	0	1	1	1	1	7	70%	0
S25	1	0.5	1	0	0.5	1	1	1	1	1	8	80%	2
S26	1	1	0	1	0	0	1	1	0.5	1	4	65%	3
S27	1	0	0	0	0	0	1	1	0	1	4	40%	0
S28	1	1	0	1	0	0	1	1	0.5	1	4	65%	0
S29	1	1	0.5	0	0.5	0.5	1	1	0.5	1	7	70%	2
S30	1	1	1	0	1	1	1	1	1	1	9	90%	2

ID	Q1	Q2	Q3	Q4	Q5	Q6	Q7	Q8	Q9	Q10	Puntuación Total	Calidad (%)	Citas
S31	1	1	1	0.5	1	1	1	1	0	1	8.5	85%	1
S32	1	0.5	1	0.5	1	1	1	1	0	1	8	80%	11
S33	1	1	0.5	0	0	0.5	0.5	0.5	0.5	1	5.5	55%	1
S34	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	2
S35	1	1	1	0.5	0	1	1	1	0	1	7.5	75%	0
S36	1	1	1	0.5	1	1	1	1	0	1	8.5	85%	0
S37	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	0
S38	0	0.5	0.5	0	1	0	0	0.5	0.5	1	4	40%	0
S39	1	1	0	0	0	0	1	1	0	1	5	50%	3
S40	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	11
S41	1	1	1	1	1	1	1	1	1	1	10	100%	4
S42	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	128
S43	1	1	1	0.5	1	0.5	1	0.5	1	1	8.5	85%	5
S44	1	1	1	0.5	0.5	1	1	0.5	0	1	7.5	75%	6
S45	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	48
S46	1	1	1	0.5	1	0.5	1	1	1	1	9	90%	25
S47	1	1	1	0.5	0.5	0.5	1	0.5	1	1	8	80%	5
S48	1	1	1	1	1	1	1	1	1	1	10	100%	0
S49	1	1	1	0	0	0	0.5	1	0	1	5.5	55%	22
S50	1	1	1	0	0	0	0.5	1	1	1	6.5	65%	3
S51	1	0.5	1	0	1	1	1	1	1	1	8.5	85%	0
S52	1	1	1	0.5	1	1	1	1	1	1	9.5	95%	0
S53	1	1	0.5	0	0.5	0	0	1	1	1	6	60%	18
S54	1	1	1	0.5	1	1	1	1	0.5	1	9	90%	15
S55	1	1	1	1	1	1	1	1	0	1	9	90%	1
S56	1	1	1	0	0.5	0	0.5	1	1	1	7	70%	1
Total												76%	

De acuerdo con los valores obtenidos y presentados en la

Tabla 80, la calidad general de los estudios incluidos es razonable, ya que la media de calidad fue del 76%.

Visión general de los artículos seleccionados

Los estudios seleccionados se publicaron entre 2001 y 2017. En la Fig. 80, se muestra el número de artículos según el año de publicación. Se puede observar un aumento en la cantidad de publicaciones en el contexto de esta revisión desde el año 2015.

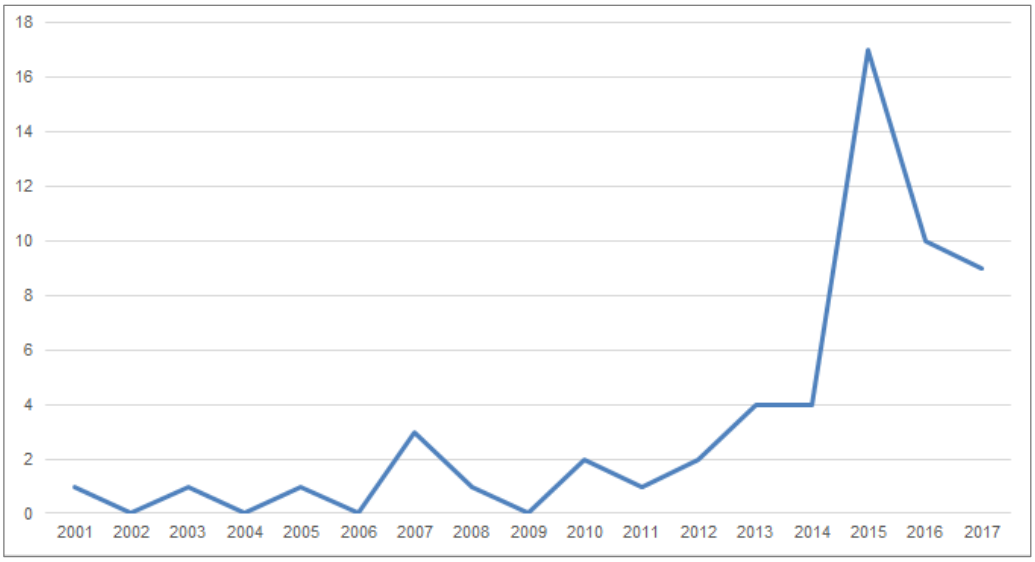


Fig. 80. Vista temporal de las publicaciones.

Luego de analizar esta visión temporal de los artículos, se puede concluir que el número de estudios sobre Pruebas Continuas y pruebas en Entrega Continua es mínimo a lo largo de los años. Aunque el número de estudios en esta área es aparentemente creciente desde el año 2013, este resultado está en línea con la afirmación de que las prácticas e investigaciones relacionadas a las pruebas en el desarrollo continuo de software han sido poco tenidas en cuenta.

Para analizar el contexto de aplicación de los estudios, se utilizaron tres categorías: industrial (empírico), académico y ambos (tanto industrial como académico). Por un lado, los estudios publicados por autores afiliados a una universidad se consideran estudios académicos. Por otro lado, los artículos que declaran explícitamente que se llevaron a cabo en una empresa real o en el lugar de trabajo de un autor en la industria, se consideran estudios industriales. Si los estudios académicos tienen experimentación o estudios de casos en entornos de trabajo reales, son clasificados como estudios académicos e industriales (ambos). Sin embargo, los artículos que tienen experimentación en laboratorios o entornos de empresas no reales se consideran solo estudios académicos.

Los resultados muestran que 16 estudios (29%) pertenecen al contexto académico. Se llevaron a cabo 13 estudios (23%) en entornos industriales y 27 estudios (48%) pertenecen a la

categoría de estudios académicos e industriales (ver Fig. 81). La mayoría de los estudios se aplicaron en la industria (71%). De aquellos artículos que se aplicaron en el contexto industrial, la mayoría de ellos también son estudios académicos. Esto puede indicar que las prácticas que están surgiendo de las universidades y de la comunidad científica, están tratando de resolver los desafíos que enfrentan las industrias. Además, también puede señalar que existe cierta aproximación entre la industria y las universidades.

Por otro lado, para analizar el método de evaluación utilizado por los artículos, se utilizaron las siguientes categorías: experimento controlado, caso de estudio, encuesta, etnografía e investigación de acción. Estas categorías para el método de evaluación fueron propuestas por Easterbrook et al. [234]. Además, se adoptaron dos categorías adicionales: "revisión sistemática de la literatura" y "no aplicable". La primera categoría se utiliza para clasificar los estudios que recopilan y analizan críticamente múltiples estudios de investigación o documentos para obtener conclusiones. La segunda categoría se refiere a los artículos que no contienen ningún tipo de método de evaluación en el estudio. Los resultados de esta clasificación se pueden ver en la Fig. 81.

La mayoría de los estudios fueron evaluados empíricamente: experimento controlado (38%) y métodos de investigación de acción (11%). 14 de los artículos fueron casos de estudio (25%). Por otro lado, también se detectaron 3 encuestas (5%), solo una RSL (2%) y 11 estudios (20%) no mencionaron ningún tipo de método de evaluación o son solo documentos de opinión. No se encontraron estudios de etnografía.

Fig. 81. Contexto de aplicación (a) y método de evaluación (b) de los artículos analizados.

Finalmente, los estudios seleccionados se clasificaron según los tipos de investigación aplicada definidos por Wieringa et al [235], como se puede ver en la Fig. 82.

El tipo de investigación más adoptado es la “solución o propuesta” con 26 estudios (46%), seguido de los artículos experimentales con 15 estudios (27%). Esto está muy relacionado con las cuestiones de investigación, ya que se buscan nuevos enfoques, soluciones, herramientas y técnicas. Por otro lado, el tipo de Investigación de Validación tiene 6 estudios (11%) y la Investigación de Evaluación tiene 5 estudios (9%). Finalmente, 4 de los estudios seleccionados (7%) pertenecen a la categoría de documentos de investigación de tipo filosófico.

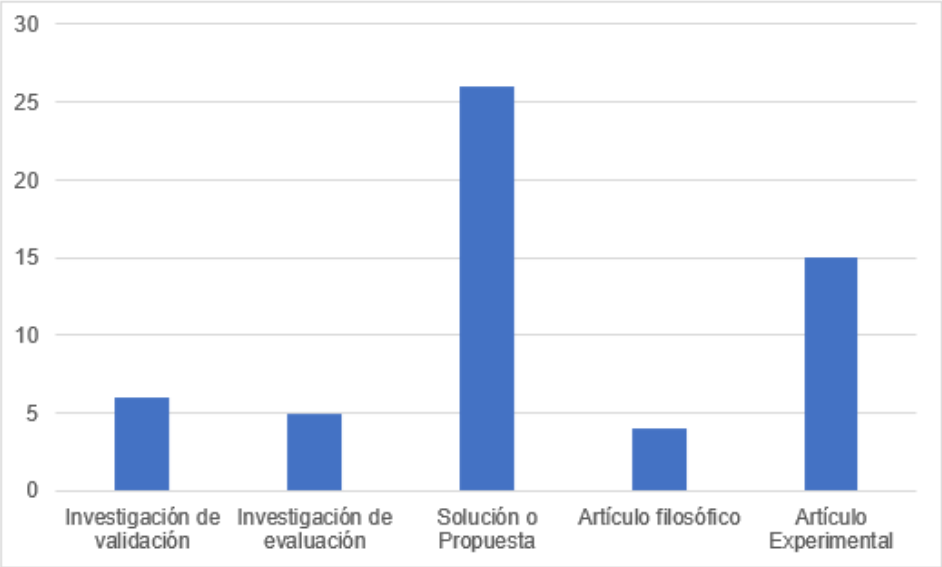


Fig. 82. Tipos de método de investigación aplicados en los estudios.

ANEXO B: Encuesta

En este anexo, se presentan los resultados correspondientes a los proyectos encuestados y el cuestionario realizado a los mismos.

Proyectos participantes

Después de un período de encuesta de cuatro meses, respondieron un total de 287 personas. Sin embargo, luego de aplicar una política para el manejo de cuestionarios inconsistentes e/o incompletos, se conservaron 255 de ellos. La política consistió en dos criterios de exclusión:

1. Rechazar resultados con cuestionarios incompletos.
2. Rechazar los resultados cuyas respuestas no son consistentes entre sí.

Los proyectos se clasificaron según el tamaño del equipo. Para ello, se utilizaron las categorías de tamaño de equipo propuestas por Yang et al. [236], basadas en las métricas de O'Connor y Yang [237].

- Equipo pequeño: menor a 16 integrantes.
- Equipo mediano: de 16 a 45 integrantes.
- Equipo grande: mayor a 45 integrantes.

La Tabla 81 muestra el número de proyectos clasificados por el tamaño del equipo.

Tabla 81. Cantidad de proyectos según su tamaño.

Tamaño	Cantidad de proyectos
Pequeño	107
Mediano	95
Grande	53

La Tabla 82 muestra el número de proyectos que trabajan en un tipo de plataforma. Es importante resaltar que la mayoría de los proyectos están trabajando en más de una plataforma.

Tabla 82. Cantidad de proyectos según la plataforma que desarrollan.

Plataforma	Cantidad de proyectos
Web GUI	138
De escritorio	22
Móvil	37
Big Data	23
Servicios Web	171
Otros	9

La Fig. 83 muestra la distribución de las plataformas que se desarrollan según el tamaño del proyecto. Como se puede ver, la mayoría de los participantes están trabajando en sitios web, tanto en la interfaz gráfica como en servicios y esto se aplica a cualquier tamaño de proyecto. También hay algunos proyectos trabajando en aplicaciones móviles y Big Data. Finalmente, se mencionaron otro tipo de plataformas más aisladas, las cuales fueron agrupados en una categoría “otros”, formada por 9 proyectos.

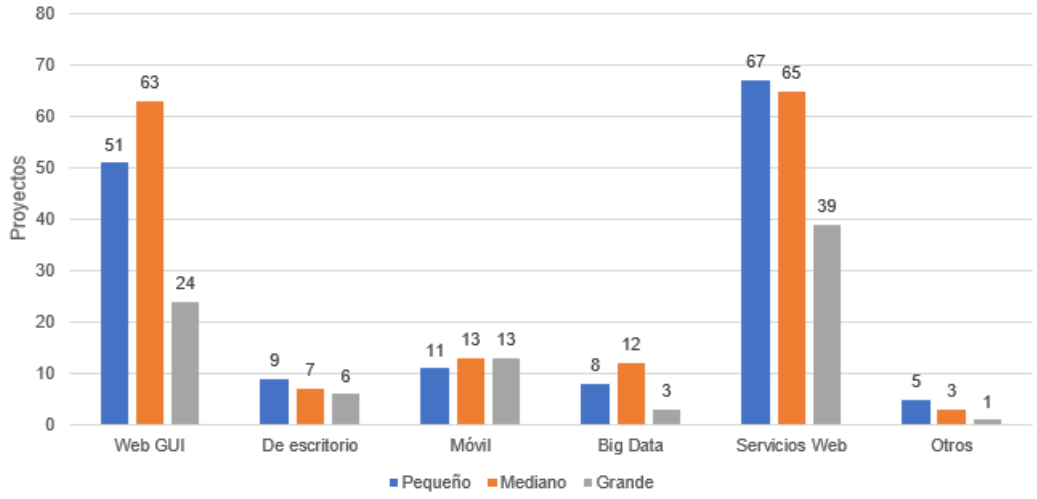


Fig. 83. Distribución de las plataformas en desarrollo según el tamaño de proyecto.

Finalmente, la Fig. 84 muestra la cantidad de proyectos según el tiempo o duración de su iteración (considerando que utilizan enfoques iterativo-incremental). La mayoría de los proyectos (65%) tienen una iteración de 2 semanas, lo que indica el uso de prácticas ágiles y lanzamientos rápidos.

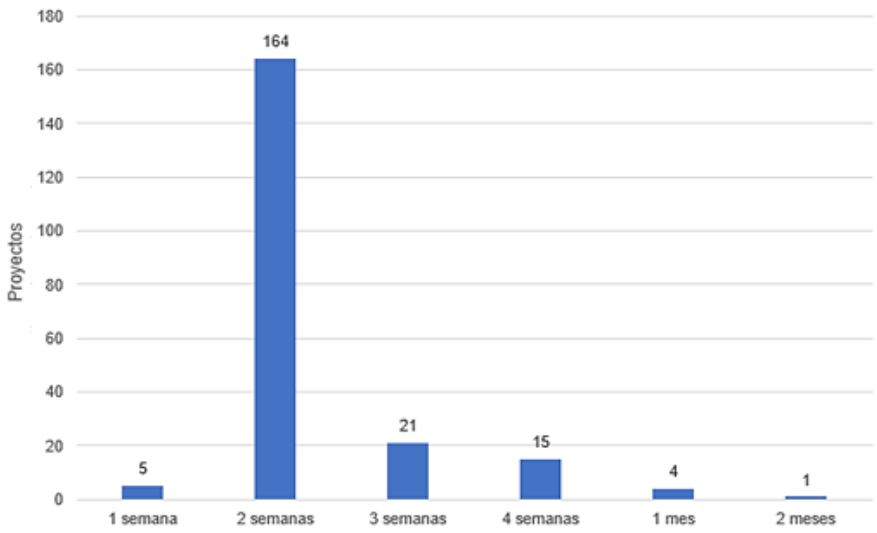


Fig. 84. Cantidad de proyectos según la duración de su iteración.

Cuestionario

En este apartado, se presenta el cuestionario realizado a las empresas que participaron de la encuesta para recabar información acerca de los problemas y soluciones en Pruebas Continuas.

Section 1: About the project

- Q1.1. Briefly describe the project being developed (e.g. web application's backend using REST web-services with Java)
- Q1.2. How many people work in the project? Please include all the members from other teams too, not only yours. The idea is to get a head count of the number of people working in the entire project.
- Q1.3. What roles do you have in your project and how many people from the previous count do you have occupying each role? (e.g. 4 software developers, 1 Scrum master, 2 manual testers, etc.) If one person is occupying more than 1 role, please mention the higher one.
- Q1.4. How long is your iteration / sprint in days? Please include only working days.

Section 2: About continuous development practices

- Q2.1. How often you build your code before integrating your changes into the mainline trunk?

a. If your previous answer was sometimes, please give some details (optional)

Q2.2. How often do you run your unit tests before integrating your changes into the mainline trunk?

a. If your previous answer was sometimes, please give some details (optional)

Q2.3. How often do you run a set of acceptance automated tests as a local regression before integrating your changes into the mainline trunk? If so, how long?

a. If your previous answer was sometimes, please give some details (optional)

Q2.4. How often do you integrate your code into the mainline trunk?

Q2.5. How long does the main automated continuous integration pipeline take to run on average in minutes?

Q2.6. How is your continuous integration flow triggered? If automated tests run on a separate process, please check an option for them too.

Q2.7. What of the following stages does your continuous integration pipeline include? (even if they are running in different processes/jobs/tasks)

Q2.8. Are you able to run the same script/s that your continuous integration server runs, in your local workstation using the command line?

Q2.9. Do you perform manual exploratory testing for new features?

Q2.10. Do you have installed the continuous integration server in a separate machine used only for that, or do you have it installed on a machine that is used for other purposes too?

Q2.11. Do you use clean environments to run your automated tests? A clean environment is an environment that does not contain any dependencies or libraries (jar files, dll files, etc.) that the application needs to run. Generally, clean environments are generated every time the tests are going to run.

Q2.12. Do you categorize your tests? (e.g. smoke, sanity, booking tests, etc.)

a. If the previous answer was "Yes", could you please provide us with some examples of the categorization you are doing.

Q2.13. Do you have a production-like environment?

Q2.14. Do developers have writing permissions for the databases in all of the environments?

a. If your previous answer was "it depends of the context", could you please give us more details?

Q2.15. If the last integration breaks the build, do you fix it immediately?

Q2.16. If your automated test suite fails in the continuous integration server, do you take care of that immediately?

Q2.17. Do you have a roll-back mechanism in case the last integration breaks the build?

Q2.18. How do you deploy your software to production?

Q2.19. What of the following assets do you store in your control version repositories? Please consider only assets that go through a control version system.

Q2.20. Do you label your software assets? For example, by using commit messages that include a clear description in every code change.

a. If your previous answer was "We label some of our software assets", could you please tell us which ones? (e.g. database schemas, base code, test scripts, deployment scripts, etc.)

Q2.21. What of the following testing types and levels do you automate?

Q2.22. What of the aforementioned automated testing types/levels are not included in your main continuous integration pipeline? If they are included but they run on a different job/task/process, please explain why.

a. If you have specified some testing types/levels in the previous question, could you please explain why? (e.g. lack of tools, lack of expertise, lack of time, etc.)

Q2.23. How often do you develop automated tests for your defects?

a. If your previous answer was sometimes, please give some details (optional)

Q2.24. What of the following attributes do you measure with your inspections or static code analysis.

Q2.25. How do you manage your test data?

Q2.26. Are your automated acceptance tests part of your definition of done? It means that a user story cannot be closed in the sprint if the automated tests for that story were not created.

Q2.27. Do your developers help testers to test the application if needed? If so, how often?

Q2.28. Do your testers help developers to ensure that acceptance criteria are met at early stages of development? (e.g. performing functional walkthroughs, working in pairs, etc.) If so, how often?

Section 3: About problems and solutions for testing in continuous development

Q3.1. Please mention all the problems, challenges or difficulties you have related to testing.

Q3.2. Please mention any kind of solutions or workarounds your team has implemented for facing problems, challenges or difficulties you had in the past or still have.

Q3.3. Feel free to add any other comments or feedback (optional)

ANEXO C: Análisis de los problemas de Pruebas Continuas encontrados

En este anexo se profundiza el análisis realizado entre los problemas de Pruebas Continuas encontrados en la literatura académica y los detectados en la industria.

En primer lugar, se formulan las siguientes hipótesis:

- H_1: La industria tiene problemas críticos relacionados con las pruebas que requieren mucho tiempo de ejecución.
- H_2: La industria tiene problemas críticos relacionados con las pruebas no deterministas.
- H_3: la industria tiene problemas críticos relacionados con resultados de pruebas ambiguos.
- H_4: La industria tiene problemas críticos relacionados con las pruebas automatizadas de la GUI.
- H_5: La industria tiene problemas críticos relacionados con las pruebas automatizadas de la interfaz de usuario web dinámica.
- H_6: La industria tiene problemas críticos relacionados con las pruebas de datos.
- H_7: La industria tiene problemas críticos relacionados con las pruebas de Big Data.
- H_8: La industria tiene problemas críticos relacionados con las pruebas automatizadas móviles.
- H_9: La industria tiene problemas críticos relacionados con las pruebas automatizadas no funcionales.
- H_{10}: La industria tiene problemas críticos relacionados con las pruebas automatizadas de aplicaciones compuestas por servicios en la nube.
- H_{11}: La industria tiene problemas relacionados con las pruebas como servicio.
- H_{12}: La industria tiene problemas relacionados con las pruebas automatizadas de servicios web.

Del mismo modo, se definieron 12 variables correspondientes con los problemas mencionados: P1, P2, P3, ..., P12, inicializadas con 0. Si un encuestado de la muestra informó un problema x, entonces la variable Px se incrementará en 1. Para validar las hipótesis, se evalúan las variables P utilizando un método de discretización global [238] y la técnica de validación utilizada en [239]. También es necesario obtener un **factor S**, que representa el número de proyectos que trabajan con una plataforma específica (por ejemplo, Big Data) o la

muestra total. De este modo, con la siguiente formula será posible identificar los problemas más importantes en Pruebas Continuas:

Puntuación (x) = Px / S

Muy alto: *0.875 < Puntuación (x) < 1*
Alto: *0.625 < Puntuación (x) < 0.875*
Medio: *0.375 < Puntuación (x) < 0.625*
Bajo: *0.125 < Puntuación (x) < 0.375*
Muy bajo: *0 < Puntuación (x) < 0.125*

Luego, H_x será cierto si la Puntuación (x) es mayor que 0.625

Como se mencionó anteriormente, se utilizaron los metadatos obtenidos de la sección de información del proyecto de la encuesta, para analizar cada problema por plataforma en desarrollo. Entonces, ese análisis se utiliza para validar las hipótesis presentadas.

H_1: La industria tiene problemas críticos relacionados con las pruebas que requieren mucho tiempo de ejecución.

Tabla 83. Problemas con pruebas que consumen mucho tiempo de ejecución, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas en pruebas que consumen mucho tiempo de ejecución
Web	126
Escritorio	19
Dispositivos Móviles	29
Big Data	1
Servicios Web	3
Sistema de Gestión de Contenido	1
Total	179

Como se muestra en la Tabla 83, las pruebas que consumen mucho tiempo de ejecución es un problema que se presenta en el desarrollo de cualquier tipo de plataforma o tecnología. De este modo, para validar **H_1**, se utilizó el total de la muestra de la encuesta como factor S.

Puntuación = 176 / 255 => Puntuación = 0.7019 (alto) => H_1 es aceptada

H_2: La industria tiene problemas críticos relacionados con las pruebas no deterministas.

Tabla 84. Problemas con pruebas no deterministas, según la plataforma en desarrollo.

Plataforma	Proyectos con pruebas no deterministas
Web	134
Escritorio	17
Dispositivos Móviles	19
Big Data	1
Servicios Web	48
Otros	5
Total	224

De acuerdo con la Tabla 84, todos los proyectos que trabajan en diferentes plataformas han informado que tienen pruebas no deterministas y, por lo tanto, para validar **H_2**, se utiliza la muestra total como factor S.

Puntuación = 224 / 255 =>	*Puntuación = 0.8784 (muy alto) =>*	*H_2 es aceptada*

H_3: la industria tiene problemas críticos relacionados con resultados de pruebas ambiguos.

Tabla 85. Problemas con resultados de pruebas ambiguos, según la plataforma en desarrollo.

Plataforma	Proyectos con resultados de pruebas ambiguos
Web	34
Escritorio	7
Dispositivos Móviles	5
Big Data	0
Servicios Web	19
Otros	0
Total	65

Con respecto a los resultados de las pruebas ambiguas, no todos los proyectos han informado de este problema (ver Tabla 85). Sin embargo, tener resultados que los

desarrolladores o cualquier otro miembro del equipo no puedan interpretar no está vinculado a un cierto tipo de tecnología. Por lo tanto, para validar **H_3** se utiliza la muestra total.

Puntuación = 65 / 255 => Puntuación = 0.2549 (bajo) => H_3 es rechazada

H_4: La industria tiene problemas críticos relacionados con las pruebas automatizadas de la GUI.

Tabla 86. Problemas con pruebas automatizadas de GUI, según la plataforma en desarrollo.

Plataforma	**Proyectos con problemas de pruebas automatizadas de GUI**
Web	92
Escritorio	17
Dispositivos Móviles	14
Big Data	0
Servicios Web	0
Sistema de Gestión de Contenido	1
Total	124

Como muestra la Tabla 86, los problemas de pruebas automatizadas de GUI están presentes en el desarrollo de plataformas que utilizan GUI: sitios web, aplicaciones de escritorio y dispositivos móviles. Además, el único proyecto que funciona en un sistema de gestión de contenido también tiene este problema. Por lo tanto, para validar **H_4**, se ha utilizado solo los proyectos que trabajan en plataformas GUI para calcular el factor S.

Factor S = Web + Escritorio + Dispositivos Móviles + Sistema de Gestión de Contenido

Factor S = 138 + 22 + 37 + 1 = 198

Puntuación = 124 / 198 => Puntuación = 0.6262 (alto) => H_4 es aceptada

H_5: La industria tiene problemas críticos relacionados con las pruebas automatizadas de la interfaz de usuario web dinámica.

Tabla 87. Problemas con pruebas automatizadas de GUI con contenido web dinámico, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas de pruebas automatizadas en aplicaciones web dinámicas
Web	47
Escritorio	0
Dispositivos Móviles	0
Big Data	0
Servicios Web	0
Otros	0
Total	47

La Tabla 87 muestra que los problemas de pruebas automatizadas de GUI con contenido web dinámico solo están presentes en el desarrollo de aplicaciones web. Por lo tanto, para validar **H_5**, se utilizan solo los proyectos que trabajan en la interfaz web para calcular el factor S.

Factor S = Web = 138

Puntuación = 124 / 198 => Puntuación = 0.3405 (bajo) => H_5 es rechazada

H_6: La industria tiene problemas críticos relacionados con las pruebas de datos.

Tabla 88. Problemas con pruebas de datos, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas de pruebas de datos
Web	12
Escritorio	2
Dispositivos Móviles	1
Big Data	1
Servicios Web	8
Otros	2
Total	26

Según la Tabla 88, todos los proyectos que trabajan en diferentes plataformas han informado que tienen problemas con los datos de prueba y, por lo tanto, para validar **H_6**, se utilizó la muestra total como factor S.

Puntuación = 65 / 255 => Puntuación = 0.1019 (muy bajo) => H_6 es rechazada

H_7: La industria tiene problemas críticos relacionados con las pruebas de Big Data.

Tabla 89. Problemas con pruebas en Big Data, según la plataforma en desarrollo.

Plataforma	**Proyectos con problemas de pruebas en Big Data**
Web	0
Escritorio	0
Dispositivos Móviles	0
Big Data	18
Servicios Web	0
Otros	0
Total	18

Como muestra la Tabla 89, es evidente que los únicos proyectos que tienen problemas con la prueba de Big Data son los que utilizan este tipo de tecnología. Por lo tanto, para validar **H_7**, se tienen en cuenta solo los proyectos que trabajan en Big Data para el cálculo del factor S.

Factor S = Big Data = 23
Puntuación = 18 / 23 => Puntuación = 0.7826 (alto) => H_7 es aceptada

H_8: La industria tiene problemas críticos relacionados con las pruebas automatizadas móviles.

Tabla 90. Problemas con pruebas en dispositivos móviles, según la plataforma en desarrollo.

Plataforma	**Proyectos con problemas de pruebas en dispositivos móviles**
Web	0
Escritorio	0

Plataforma	Proyectos con problemas de pruebas en dispositivos móviles
Dispositivos Móviles	17
Big Data	0
Servicios Web	0
Otros	0
Total	17

De manera similar a los problemas de Big Data, los únicos proyectos que tienen problemas para realizar pruebas en dispositivos móviles son los que funcionan con este tipo de tecnología (ver Tabla 90). Por lo tanto, para validar H8, se han utilizado solo los proyectos que trabajan en plataformas móviles para calcular el factor S.

Factor S = Dispositivos móviles = 37

Puntuación = 17 / 37 => Puntuación = 0.4594 (medio) => H_8 es rechazada

H_9: La industria tiene problemas críticos relacionados con las pruebas automatizadas no funcionales.

Tabla 91. Problemas con pruebas automatizadas no funcionales, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas de pruebas automatizadas no funcionales
Web	12
Escritorio	2
Dispositivos Móviles	2
Big Data	1
Servicios Web	4
Otros	2
Total	23

De acuerdo con la Tabla 91, todos los proyectos que trabajan en diferentes plataformas han informado que tienen problemas para ejecutar pruebas automatizadas no funcionales y, por lo tanto, para validar **H_9**, se utilizó la muestra total como factor S.

Puntuación = 23 / 255 => Puntuación = 0.0901 (muy bajo) => H_9 es rechazada

H_{10}: La industria tiene problemas críticos relacionados con las pruebas automatizadas de aplicaciones compuestas por servicios en la nube.

Tabla 92. Problemas con pruebas automatizadas de aplicaciones compuestas por servicios en la nube, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas automatizadas de aplicaciones compuestas por servicios en la nube
Web	1
Escritorio	0
Dispositivos Móviles	17
Big Data	0
Servicios Web	0
Sistema de seguimiento de incidentes en línea	1
Total	2

Como muestra la Tabla 92, solo dos proyectos informaron que las pruebas automatizadas de aplicaciones compuestas por servicios en la nube eran un problema. Uno funciona en una aplicación web y el otro está desarrollando un sistema de seguimiento de incidentes en línea compuesto por servicios en la nube. Por lo tanto, para validar **H_{10}** y calcular el factor S, se ha utilizado solo los proyectos que trabajan en aplicaciones web y el que trabaja en el sistema de seguimiento de incidentes en línea.

Factor S = Web + Sistema de seguimiento de incidentes en línea

Factor S = 138 + 1 = 139

Puntuación = 2 / 139 => Puntuación = 0.0143 (muy bajo) => H_{10} es rechazada

H_{11}: La industria tiene problemas relacionados con las pruebas como servicio.

Tabla 93. Problemas en TaaS, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas en TaaS
Web	0
Escritorio	0
Dispositivos Móviles	0
Big Data	0
Servicios Web	0
Otros	0
Total	0

Ninguno de los que respondieron ha reportado problemas con TaaS (ver la Tabla 93). Por lo tanto, la Puntuación de **H_{11}** es 0.

Puntuación = 0 => H_{11} es rechazada

H_{12}: La industria tiene problemas relacionados con las pruebas automatizadas de servicios web.

Tabla 94. Problemas en pruebas automatizadas de servicios web, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas en pruebas de servicios web
Web	0
Escritorio	0
Dispositivos Móviles	0
Big Data	0
Servicios Web	0
Otros	0
Total	0

De manera similar a los problemas de Big Data y las pruebas automatizadas para dispositivos móviles, los únicos proyectos que tienen problemas con las pruebas automatizadas de servicios web son los que funcionan con este tipo de tecnología (ver Tabla 94). Por lo tanto,

para validar **H_{12}**, se ha utilizado solo los proyectos que trabajan en servicios web para calcular el factor S.

Factor S = Servicios Web = 171
Puntuación = 58 / 171 => Puntuación = 0.3391 (bajo) => H_{12} es rechazada

Otros problemas no contemplados en las hipótesis

Además de los problemas estudiados y validados a través de las hipótesis, las compañías informaron otros problemas o desafíos que podrían afectar la implementación de un proceso de pruebas en un entorno de desarrollo continuo. Esos problemas son los entornos inestables y la falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas exclusivos para entornos de desarrollo continuo. La Tabla 95 muestra su relación con la plataforma que se está desarrollando.

Tabla 95. Problemas de ambientes inestables y falta de procesos para pruebas en entornos continuos, según la plataforma en desarrollo.

Plataforma	Proyectos con problemas de ambientes inestables	Proyectos con falta de procesos para pruebas automatizadas en entornos de desarrollo continuo
Web	18	46
Escritorio	4	16
Dispositivos Móviles	6	12
Big Data	1	15
Servicios Web	7	20
Otros	0	3
Total	36	112

Usando la misma fórmula, se puede determinar la severidad de estos problemas de la siguiente manera:

Ambientes inestables	*=>*	*Puntuación = 36 / 255 = 0.1411*	*=>*	*Bajo*
Falta de procesos	*=>*	*Puntuación = 112 / 255 = 0.4392*	*=>*	*Medio*

Finalmente, la Tabla 96 representa la relación entre los problemas encontrados en la literatura y los encontrados con esta encuesta.

Tabla 96. Relación entre los problemas encontrados en la literatura y la industria con su severidad.

Problema	**Literatura**	**Industria**	**Severidad**
Pruebas que consumen mucho tiempo de ejecución.	Presente	Presente	Alta (H_1 aceptada)
Pruebas no deterministas.	Presente	Presente	Muy alta (H_2 aceptada)
Resultados de ejecución de pruebas ambiguos.	Presente	Presente	Baja (H_3 rechazada)
Pruebas automatizadas de interfaz gráfica de usuario.	Presente	Presente	Alta (H_4 aceptada)
Pruebas automatizadas de web con contenido dinámico.	Presente	Presente	Baja (H_5 rechazada)
Pruebas de datos.	Presente	Presente	Muy baja (H_6 rechazada)
Pruebas en Big Data.	Presente	Presente	Alta (H_7 aceptada)
Pruebas en dispositivos móviles.	Presente	Presente	Media (H_8 rechazada)
Pruebas automatizadas de requerimientos no funcionales.	Presente	Presente	Muy baja (H_9 rechazada)
Pruebas de aplicaciones compuestas por servicios en la nube.	Presente	Presente	Muy baja (H_{10} rechazada)
Pruebas como un servicio.	Presente	Faltante	Muy baja (H_{11} rechazada)
Pruebas de servicios web.	Presente	Presente	Baja (H_{12} rechazada)
Ambientes inestables.	Faltante	Presente	Muy baja
Falta de procedimientos, patrones y buenas prácticas para pruebas automatizadas en desarrollo continuo.	Faltante	Presente	Media

REFERENCIAS

[1] B. Boehm, «A view of 20th and 21st century software engineering», en *Proceedings of the 28th international conference on Software engineering*, New York, NY, USA, may 2006, pp. 12-29, doi: 10.1145/1134285.1134288.

[2] G. Tassey, *The Economic Impacts of Inadequate Infrastructure for Software Testing: Final Report*, Edición: Spi. Place of publication not identified: Diane Pub Co, 2003.

[3] M. Cohn, *Succeeding with agile: Software development using scrum*, Edición: 01. Upper Saddle River, NJ: Addison-Wesley Educational Publishers Inc, 2009.

[4] M. Fowler, «Continuous Integration», *martinfowler.com*. https://martinfowler.com/articles/continuousIntegration.html (accedido jun. 02, 2018).

[5] C. Rossi, E. Shibley, S. Su, K. Beck, T. Savor, y M. Stumm, «Continuous Deployment of Mobile Software at Facebook (Showcase)», nov. 2016, Accedido: jul. 09, 2018. [En línea]. Disponible en: https://research.fb.com/publications/continuous-deployment-of-mobile-software-at-facebook-showcase.

[6] P. Rodríguez *et al.*, «Continuous deployment of software intensive products and services: A systematic mapping study», *J. Syst. Softw.*, vol. 123, pp. 263-291, ene. 2017, doi: 10.1016/j.jss.2015.12.015.

[7] J. Humble y D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*, Edición: 01. Upper Saddle River, NJ: Pearson Professional, 2010.

[8] M. A. Mascheroni, M. K. Cogliolo, y E. Irrazábal, «Automatización de pruebas de compatibilidad web en un entorno de desarrollo continuo de software», presentado en Simposio Argentino de Ingeniería de Software (ASSE 2016) - JAIIO 45 (Tres de Febrero, 2016)., nov. 2016, Accedido: jun. 28, 2018. [En línea]. Disponible en: http://hdl.handle.net/10915/57081.

[9] P. Duvall, S. Matyas, y A. Glover, *Continuous integration: Improving software quality and reducing risk*, Edición: 01. Upper Saddle River, NJ: Pearson Professional, 2007.

[10] M. Fowler, «Continuous Delivery», *martinfowler.com*. https://martinfowler.com/bliki/ContinuousDelivery.html (accedido jun. 02, 2018).

[11] M. A. Mascheroni y E. Irrazábal, «Continuous Testing and Solutions for Testing Problems in Continuous Delivery: A Systematic Literature Review», *Comput. Sist.*, vol. 22, n.º 3, sep. 2018, doi: 10.13053/cys-22-3-2794.

[12] E. Laukkanen, J. Itkonen, y C. Lassenius, «Problems, causes and solutions when adopting continuous delivery — A systematic literature review», *Inf. Softw. Technol.*, vol. 82, pp. 55-79, feb. 2017, doi: 10.1016/j.infsof.2016.10.001.

[13] L. Chen, «Continuous Delivery: Huge Benefits, but Challenges Too», *IEEE Softw.*, vol. 32, n.º 2, pp. 50-54, mar. 2015, doi: 10.1109/MS.2015.27.

[14] M. A. Mascheroni y E. Irrazábal, «Problemas que afectan a la calidad de software en entrega continua y pruebas continuas», presentado en XXIV Congreso Argentino de Ciencias de la Computación (La Plata, 2018)., 2018, Accedido: abr. 05, 2019. [En línea]. Disponible en: http://sedici.unlp.edu.ar/handle/10915/73270.

[15] M. A. Mascheroni, E. Irrazábal, J. A. Carruthers, y J. A. Pinto, «Rapid Releases and Testing Problems at the industry: A survey», presentado en XXV Congreso Argentino de Ciencias de la Computación (CACIC) (Universidad Nacional de Río Cuarto, Córdoba, 14 al 18 de octubre de 2019), 2019, Accedido: may 23, 2020. [En línea]. Disponible en: http://sedici.unlp.edu.ar/handle/10915/91103.

[16] O. Prusak, «Continuous testing: The missing link in the continuous delivery process». https://www.blazemeter.com/blog/continuous-testing-missing-link-continuous-delivery-process (accedido jul. 04, 2018).

[17] BlazeMeter, «Continuous Testing in practice. Completing the Continuous Delivery Process». 2015, [En línea]. Disponible en: https://info.blazemeter.com/shift-left-testing.

[18] C. Vassallo *et al.*, «Continuous Delivery Practices in a Large Financial Organization», en *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, oct. 2016, pp. 519-528, doi: 10.1109/ICSME.2016.72.

[19] «Automated Testing for Continuous Delivery Pipelines», *PNSQC*, jul. 22, 2016. https://www.pnsqc.org/automated-testing-continuous-delivery-pipelines/ (accedido abr. 11, 2020).

[20] M. A. Mascheroni y E. Irrazábal, «Identifying Key Success Factors in Stopping Flaky Tests in Automated REST Service Testing», *J. Comput. Sci. Technol.*, vol. 18, n.º 2, oct. 2018, Accedido: ago. 31, 2019. [En línea]. Disponible en: http://sedici.unlp.edu.ar/handle/10915/70119.

[21] D. Ståhl y J. Bosch, «Automated software integration flows in industry: A multiple-case study», en *Companion Proceedings of the 36th International Conference on Software Engineering*, New York, NY, USA, 2014, pp. 54-63, doi: 10.1145/2591062.2591186.

[22] A. Debbiche, M. Dienér, y R. B. Svensson, «Challenges when adopting continuous integration: A case study», en *Product-Focused Software Process Improvement*, dic. 2014, pp. 17-32, doi: 10.1007/978-3-319-13835-0_2.

[23] B. Fitzgerald y K.-J. Stol, «Continuous software engineering and beyond: Trends and challenges», en *Proceedings of the 1st International Workshop on Rapid Continuous Software Engineering*, New York, NY, USA, 2014, pp. 1-9, doi: 10.1145/2593812.2593813.

[24] G. G. Claps, R. Berntsson Svensson, y A. Aurum, «On the journey to continuous deployment: Technical and social challenges along the way», *Inf. Softw. Technol.*, vol. 57, pp. 21-31, ene. 2015, doi: 10.1016/j.infsof.2014.07.009.

[25] B. Fitzgerald y K.-J. Stol, «Continuous software engineering: A roadmap and agenda», *J. Syst. Softw.*, vol. 123, pp. 176-189, ene. 2017, doi: 10.1016/j.jss.2015.06.063.

[26] M. V. Mäntylä, B. Adams, F. Khomh, E. Engström, y K. Petersen, «On rapid releases and software testing: a case study and a semi-systematic literature review», *Empir. Softw. Eng.*, vol. 20, n.º 5, pp. 1384-1425, oct. 2015, doi: 10.1007/s10664-014-9338-4.

[27] S. Neely y S. Stolt, «Continuous Delivery? Easy! Just change everything (well, maybe it is not that easy)», en *2013 Agile Conference*, ago. 2013, pp. 121-128, doi: 10.1109/AGILE.2013.17.

[28] IEEE Computer Society, «Software Engineering Body of Knowledge (SWEBOK)», 2004. https://www.computer.org/education/bodies-of-knowledge/software-engineering (accedido jun. 09, 2019).

[29] E. Marcos y A. Marcos, «An aristotelian approach to the methodological research: A method for data models construction», en *Information Systems: The Next Generation*, University of York, 1999, pp. 532-543.

[30] E. Marcos, «Investigación en ingeniería del software vs. desarrollo software», presentado en MIFISIS, 2002.

[31] M. Bunge, *La investigación científica: su estrategia y su filosofía*. Siglo XXI, 2000.

[32] E. Schön, «How Do Agile Practices Support Organizing a Ph.D.?», *IT Prof.*, vol. 20, n.º 6, pp. 82-86, nov. 2018, doi: 10.1109/MITP.2018.2876927.

[33] B. Kitchenham, «Procedures for Performing Systematic Reviews», *Keele UK Keele Univ*, vol. 33, ago. 2004.

[34] F. J. F. Jr, *Survey Research Methods*. SAGE Publications, 2013.

[35] J. Biolchini, P. Gomes Mian, A. Candida, y C. Natali, «Systematic Review in Software Engineering», ene. 2005.

[36] M. G. Bocco, J. A. C. Lemus, y M. G. P. Velthuis, *Métodos de investigación en ingeniería del software*. Ediciones de la U, 2014.

[37] S. L. Pfleeger y B. A. Kitchenham, «Principles of Survey Research: Part 1: Turning Lemons into Lemonade», *SIGSOFT Softw Eng Notes*, vol. 26, n.º 6, pp. 16-18, nov. 2001, doi: 10.1145/505532.505535.

[38] B. A. Kitchenham y S. L. Pfleeger, «Personal Opinion Surveys», en *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, y D. I. K. Sjøberg, Eds. London: Springer London, 2008, pp. 63-92.

[39] K. Lewin, «Frontiers in Group Dynamics: II. Channels of Group Life; Social Planning and Action Research», *Hum. Relat.*, vol. 1, n.º 2, pp. 143-153, nov. 1947, doi: 10.1177/001872674700100201.

[40] A. T. Wood-Harper, «Research Methods in Information Systems: Using Action Research», en *Research Methods in Information Systems*, pp. 169-191.

[41] C. B. Seaman, «Qualitative methods in empirical studies of software engineering», *IEEE Trans. Softw. Eng.*, vol. 25, n.º 4, pp. 557-572, jul. 1999, doi: 10.1109/32.799955.

[42] F. J. Pino, M. Piattini, y G. Horta Travassos, «Managing and developing distributed research projects in software engineering by means of action-research», *Rev. Fac. Ing. Univ. Antioquia*, n.º 68, pp. 61-74, sep. 2013.

[43] «Principles for Participatory Action Research - Robin McTaggart, 1991». https://journals.sagepub.com/doi/abs/10.1177/0001848191041003003 (accedido ago. 12, 2020).

[44] N. Kock y F. Lau, «Information systems action research: serving two demanding masters», *Inf. Technol. People*, vol. 14, n.º 1, ene. 2001, doi: 10.1108/itp.2001.16114aaa.001.

[45] Y. Wadsworth, «What is participatory Action Research? - Paper 2», *Action research international*, 1998.

[46] D. L. Parnas, «Software Fundamentals», D. M. Hoffman y D. M. Weiss, Eds. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 2001, pp. 257-266.

[47] B. Kitchenham y S. L. Pfleeger, «Software quality: the elusive target», *IEEE Softw.*, vol. 13, n.º 1, pp. 12-21, ene. 1996, doi: 10.1109/52.476281.

[48] D. A. Garvin, «What Does “Product Quality” Really Mean?», *MIT Sloan Management Review*. https://sloanreview.mit.edu/article/what-does-product-quality-really-mean/ (accedido jul. 25, 2019).

[49] C. Robson, *Real World Research*, 3rd edition. Chichester: John Wiley & Sons, 1911.

[50] «ISO 25000 STANDARDS». https://iso25000.com/index.php/en/iso-25000-standards (accedido jul. 25, 2019).

[51] «ISO 25010». http://iso25000.com/index.php/en/iso-25000-standards/iso-25010 (accedido jun. 09, 2019).

[52] «Manifesto for Agile Software Development». https://agilemanifesto.org/ (accedido may 26, 2019).

[53] V. Stavridou, «Integration in software intensive systems», *J. Syst. Softw.*, vol. 48, n.º 2, pp. 91-104, oct. 1999, doi: 10.1016/S0164-1212(99)00049-7.

[54] T. Chow y D.-B. Cao, «A survey study of critical success factors in agile software projects», *J. Syst. Softw.*, vol. 81, n.º 6, pp. 961-971, jun. 2008, doi: 10.1016/j.jss.2007.08.020.

[55] G. Booch, *Object Solutions: Managing the Object-Oriented Project*, Edición: New. Menlo Park, Ca: Benjamin Cummings, 1995.

[56] M. A. Cusumano, W. S. Richard, y R. W. Selby, *Microsoft secrets: How the world's most powerful software company creates technology, shapes markets, and manages people: how the world's most technology, shapes markets and manages people*, Edición: 1st Touchstone Ed. New York: Touchstone, 1988.

[57] S. C. McConnell, *Software project survival guide: How to be sure your first important project isn't your last*. Microsoft Press,U.S., 1997.

[58] N. Vafaie y M. Arvidsson, «Metrics to mesure the impact of continuous integration- An empirical study», may 2015, Accedido: jun. 02, 2018. [En línea]. Disponible en: https://gupea.ub.gu.se/handle/2077/38845.

[59] P. Clements y L. Northrop, *Software Product Lines: Practices and Patterns: Practices and Patterns*, Edición: 01. Boston San Francisco New York Toronto Montreal London Munich Paris Madrid Capetown Sydney Tokyo Singapore Mexico City: Pearson Professional, 2015.

[60] E. Irrazábal y J. Garzás, «Análisis de métricas básicas y herramientas de código libre para medir la mantenibilidad», *REICIS Rev. Esp. Innov. Calid. E Ing. Softw.*, vol. 6, n.º 3, 2010, Accedido: jun. 09, 2018. [En línea]. Disponible en: http://www.redalyc.org/resumen.oa?id=92218768005.

[61] T. Dybå y T. Dingsøyr, «Empirical studies of agile software development: A systematic review», *Inf. Softw. Technol.*, vol. 50, n.º 9, pp. 833-859, ago. 2008, doi: 10.1016/j.infsof.2008.01.006.

[62] R. C. Martin, *Agile Software Development, Principles, Patterns, and Practices*, 1st edition. Upper Saddle River, N.J: Pearson, 2002.

[63] M. Poppendieck y T. Poppendieck, *Lean Software Development: An Agile Toolkit*. Boston: Addison-Wesley Professional, 2003.

[64] L. Bocock y A. Martin, «There's Something about Lean: A Case Study», en *2011 Agile Conference*, ago. 2011, pp. 10-19, doi: 10.1109/AGILE.2011.44.

[65] P. Middleton, «Lean Software Development: Two Case Studies», *Softw. Qual. J.*, vol. 9, n.º 4, pp. 241-252, dic. 2001, doi: 10.1023/A:1013754402981.

[66] A. L. Alwardt, N. Mikeska, R. J. Pandorf, y P. R. Tarpley, «A lean approach to designing for software testability», en *2009 IEEE AUTOTESTCON*, sep. 2009, pp. 178-183, doi: 10.1109/AUTEST.2009.5314039.

[67] «What Is Web 2.0». https://oreilly.com{file} (accedido may 20, 2019).

[68] P. Agarwal, «Continuous SCRUM: Agile Management of SAAS Products», en *Proceedings of the 4th India Software Engineering Conference*, New York, NY, USA, 2011, pp. 51-60, doi: 10.1145/1953355.1953362.

[69] D. G. Feitelson, E. Frachtenberg, y K. L. Beck, «Development and Deployment at Facebook», *IEEE Internet Comput.*, vol. 17, n.º 4, pp. 8-17, jul. 2013, doi: 10.1109/MIC.2013.25.

[70] F. J. Lacoste, «Killing the Gatekeeper: Introducing a Continuous Integration System», en *2009 Agile Conference*, ago. 2009, pp. 387-392, doi: 10.1109/AGILE.2009.35.

[71] T. van der Storm, «Backtracking Incremental Continuous Integration», en *2008 12th European Conference on Software Maintenance and Reengineering*, abr. 2008, pp. 233-242, doi: 10.1109/CSMR.2008.4493318.

[72] Standish Group, «Chaos Report 2013». 2013.

[73] E. Ries, *The Lean Startup: How Today's Entrepreneurs Use Continuous Innovation to Create Radically Successful Businesses*, 1 edition. New York: Currency, 2011.

[74] M. Fowler, «Software Development in the 21st century», presentado en ThoughtWorks XCONF 2014, Hamburg & Manchester, 2014.

[75] «Continuous Delivery, Deployment & Integration: 20 Key Differences», *Stackify*, jul. 25, 2017. https://stackify.com/continuous-delivery-vs-continuous-deployment-vs-continuous-integration/ (accedido may 26, 2019).

[76] A. Phillips, M. Sens, A. de Jonge, y M. van Holsteijn, *The IT Manager's Guide to Continuous Delivery: Delivering Software in Days*, 1 edition. BookBaby, 2014.

[77] M. Shahin, M. A. Babar, y L. Zhu, «Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices», *IEEE Access*, vol. 5, pp. 3909-3943, 2017, doi: 10.1109/ACCESS.2017.2685629.

[78] J. Gregory y L. Crispin, *More Agile Testing: Learning Journeys for the Whole Team*, 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2014.

[79] L. Crispin y J. Gregory, *Agile Testing: A Practical Guide for Testers and Agile Teams*, 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2009.

[80] «ISO 25012». http://iso25000.com/index.php/en/iso-25000-standards/iso-25012 (accedido jun. 09, 2019).

[81] «ISO 25040». http://iso25000.com/index.php/en/iso-25000-standards/iso-25040 (accedido jun. 09, 2019).

[82] D. Graham, E. V. Veenendaal, I. Evans, y R. Black, *Foundations of Software Testing: ISTQB Certification*, Revised edition. Cengage Learning Emea, 2008.

[83] R. Black *et al.*, *Agile Testing Foundations: An ISTQB Foundation Level Agile Tester Guide*. BCS, The Chartered Institute for IT, 2017.

[84] P. Louridas, «Static code analysis», *IEEE Softw.*, vol. 23, n.º 4, pp. 58-61, jul. 2006, doi: 10.1109/MS.2006.114.

[85] B. A. Wichmann, A. A. Canning, D. L. Clutterbuck, L. A. Winsborrow, N. J. Ward, y D. W. R. Marsh, «Industrial perspective on static analysis», *Softw. Eng. J.*, vol. 10, n.º 2, pp. 69-75, mar. 1995, doi: 10.1049/sej.1995.0010.

[86] E. Irrazábal, «Mejora de la mantenibilidad con un modelo de medición de la calidad: resultados en una gran empresa», presentado en XXI Congreso Argentino de Ciencias de la Computación, Junín, Buenos Aires, 2015, Accedido: jun. 09, 2019. [En línea]. Disponible en: http://sedici.unlp.edu.ar/handle/10915/50334.

[87] F. A. Fontana, R. Roveda, y M. Zanoni, «Tool Support for Evaluating Architectural Debt of an Existing System: An Experience Report», en *Proceedings of the 31st Annual ACM Symposium on Applied Computing*, New York, NY, USA, 2016, pp. 1347-1349, doi: 10.1145/2851613.2851963.

[88] «IEEE 1008-1987 - IEEE Standard for Software Unit Testing». https://standards.ieee.org/standard/1008-1987.html (accedido jun. 15, 2019).

[89] M. Rodriguez, M. Piattini, y C. Ebert, «Software Verification and Validation Technologies and Tools», *IEEE Softw.*, vol. 36, n.º 2, pp. 13-24, mar. 2019, doi: 10.1109/MS.2018.2883354.

[90] C. Jöngren, *Automated Integration Testing: An Evaluation of CruiseControl.NET*. Skolan för datavetenskap och kommunikation, Kungliga Tekniska högskolan, 2008.

[91] «The Practical Test Pyramid», *martinfowler.com*. https://martinfowler.com/articles/practical-test-pyramid.html (accedido jun. 17, 2019).

[92] «ISO/IEC 19501:2005», *ISO*. http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/26/32620.html (accedido jun. 17, 2019).

[93] P. Sabev y K. Grigorova, «A Comparative Study of GUI Automated Tools for Software Testing», abr. 2017.

[94] L. Chen, «Continuous Delivery: Overcoming adoption challenges», *J. Syst. Softw.*, vol. 128, pp. 72-86, jun. 2017, doi: 10.1016/j.jss.2017.02.013.

[95] M. T. Nygard, *Release It!: Design and Deploy Production-Ready Software*, Edición: 1. Raleigh, N.C: Pragmatic Bookshelf, 2017.

[96] F. Maila-Maila, M. Intriago-Pazmiño, y J. Ibarra-Fiallo, «Evaluation of Open Source Software for Testing Performance of Web Applications», en *New Knowledge in Information Systems and Technologies*, 2019, pp. 75-82.

[97] J. Bach, «Exploratory Testing», *Satisfice, Inc.* https://www.satisfice.com/exploratory-testing (accedido jul. 25, 2019).

[98] J. A. Whittaker, *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*, 1 edition. Upper Saddle River, NJ: Addison-Wesley Professional, 2009.

[99] «ISO 9241-11:2018(en), Ergonomics of human-system interaction — Part 11: Usability: Definitions and concepts». https://www.iso.org/obp/ui/#iso:std:iso:9241:-11:ed-2:v1:en (accedido jul. 25, 2019).

[100] M. A. Mascheroni, C. L. Greiner, G. N. Dapozo, y M. G. Estayno, «Herramienta para automatizar la evaluación de la usabilidad en productos software», presentado en XVIII Congreso Argentino de Ciencias de la Computación, oct. 2012, Accedido: jul. 25, 2019. [En línea]. Disponible en: http://hdl.handle.net/10915/23732.

[101] M. A. Mascheroni, C. L. Greiner, G. N. Dapozo, y M. G. Estayno, «Automatización de la evaluación de la usabilidad del software», presentado en XV Workshop de Investigadores en Ciencias de la Computación, jun. 2013, Accedido: jul. 25, 2019. [En línea]. Disponible en: http://hdl.handle.net/10915/27243.

[102] M. A. Mascheroni, C. L. Greiner, R. H. Petris, G. N. Dapozo, y M. G. Estayno, «Calidad de software e ingeniería de usabilidad», presentado en XIV Workshop de Investigadores

en Ciencias de la Computación, 2012, Accedido: jul. 25, 2019. [En línea]. Disponible en: http://hdl.handle.net/10915/19202.

[103] W. S. Humphrey, *Managing the Software Process*. Reading, Mass: Addison-Wesley Professional, 1989.

[104] A. Mitasiunas, T. Rout, R. V. O'Connor, y A. Dorling, *Software Process Improvement and Capability Determination: 14th International Conference, SPICE 2014, Vilnius, Lithuania, November 4-6, 2014. Proceedings*. Springer, 2014.

[105] «TMMi Model – TMMI». https://www.tmmi.org/tmmi-model/ (accedido ago. 25, 2019).

[106] C. Garcia, A. Dávila, y M. Pessoa, «Test Process Models: Systematic Literature Review», en *Software Process Improvement and Capability Determination*, 2014, pp. 84-93.

[107] «CMMI Institute - CMMI Development». https://cmmiinstitute.com/cmmi/dev (accedido ago. 24, 2019).

[108] jgarzas, «Otros modelos de calidad software: Modelos para pruebas», *Javier Garzas*, ene. 22, 2009. https://www.javiergarzas.com/2009/01/otros-modelos-de-calidad-software.html (accedido ago. 24, 2019).

[109] A. Pope, *An Essay on Criticism*. 1711.

[110] V. E. Jyothi, R. Krishna, y N. Rao, «Agility in web application development–success embraces at the enterprise level», *International Journal of Current Engineering and Technology*, vol. 6, n.º 3, pp. 1049-1053, 2016.

[111] D. R. Kuhn, D. R. Wallace, y A. M. Gallo, «Software fault interactions and implications for software testing», *IEEE Trans. Softw. Eng.*, vol. 30, n.º 6, pp. 418-421, jun. 2004, doi: 10.1109/TSE.2004.24.

[112] J.-L. Lions, «Ariane 5, Flight 501, Report of the Inquiry Board», 1996.

[113] «List of software bugs», *Wikipedia*. may 30, 2018, Accedido: jun. 27, 2018. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=List_of_software_bugs&oldid=843571754.

[114] G. Brooks, «Team pace keeping build times down», en *Agile 2008 Conference*, ago. 2008, pp. 294-297, doi: 10.1109/Agile.2008.41.

[115] E. Laukkanen, T. O. A. Lehtinen, J. Itkonen, M. Paasivaara, y C. Lassenius, «Bottom-up adoption of continuous delivery in a stage-gate managed software organization», en *Proceedings of the 10th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2016, p. 45:1-45:10, doi: 10.1145/2961111.2962608.

[116] F. Cannizzo, R. Clutton, y R. Ramesh, «Pushing the boundaries of testing and continuous integration», en *Agile 2008 Conference*, ago. 2008, pp. 501-505, doi: 10.1109/Agile.2008.31.

[117] M. Leppänen *et al.*, «The highways and country roads to continuous deployment», *IEEE Softw.*, vol. 32, n.º 2, pp. 64-72, mar. 2015, doi: 10.1109/MS.2015.50.

[118] A. Debbiche y M. Dienér, «Assessing challenges of continuous integration in the context of software requirements breakdown: A case study», Tesis de Maestría, University of Gothenburg, Gothenburg, Suecia, 2014.

[119] E. Alégroth, R. Feldt, y L. Ryrholm, «Visual GUI testing in practice: challenges, problems and limitations», *Empir. Softw. Eng.*, vol. 20, n.º 3, pp. 694-744, jun. 2015, doi: 10.1007/s10664-013-9293-5.

[120] E. Borjesson y R. Feldt, «Automated system testing using visual gui testing tools: A comparative study in industry», en *Verification and Validation 2012 IEEE Fifth International Conference on Software Testing*, abr. 2012, pp. 350-359, doi: 10.1109/ICST.2012.115.

[121] L. Pradhan, «User interface test automation and its challenges in an industrial scenario», Tesis de Maestría, School of Innovation, Design and Engineering. Mälardalen University, Suecia, 2012.

[122] P. Suwala, «Challenges with modern web testing», Tesis de Maestría, Institutionen för datavetenskap. Department of Computer and Information Science. Linköpings Universitet, Linköping, Suecia, 2015.

[123] M. A. Mascheroni, M. K. Cogliolo, y E. Irrazábal, «Automatic detection of Web Incompatibilities using Digital Image Processing», *Electron. J. SADIO EJS*, vol. 16, pp. 29-45, sep. 2017.

[124] L. N. Sabaren, M. A. Mascheroni, C. L. Greiner, y E. Irrazábal, «A Systematic Literature Review in Cross-browser Testing», *J. Comput. Sci. Technol.*, vol. 18, n.º 01, pp. e03-e03, abr. 2018, doi: 10.24215/16666038.18.e03.

[125] L. Sabaren, M. A. Mascheroni, C. L. Greiner, y E. Irrazábal, «Una revisión sistemática de la literatura en pruebas de compatibilidad web», presentado en XXIII Congreso Argentino de Ciencias de la Computación (La Plata, 2017)., oct. 2017, Accedido: ago. 31, 2019. [En línea]. Disponible en: http://sedici.unlp.edu.ar/handle/10915/63811.

[126] D. Huizinga y A. Kolawa, *Automated defect prevention: Best practices in software management*, Edición: 1. Hoboken, N.J: John Wiley & Sons, 2007.

[127] N. Garg, S. Singla, y S. Jangra, «Challenges and techniques for testing of big data», *Procedia Comput. Sci.*, vol. 85, pp. 940-948, ene. 2016, doi: 10.1016/j.procs.2016.05.285.

[128] K. Muşlu, Y. Brun, y A. Meliou, «Data debugging with continuous testing», en *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, New York, NY, USA, 2013, pp. 631-634, doi: 10.1145/2491411.2494580.

[129] H. Muccini, A. D. Francesco, y P. Esposito, «Software testing of mobile applications: Challenges and future research directions», en *2012 7th International Workshop on Automation of Software Test (AST)*, jun. 2012, pp. 29-35, doi: 10.1109/IWAST.2012.6228987.

[130] D. Ståhl y J. Bosch, «Experienced Benefits of Continuous Integration in Industry Software Product Development: A Case Study», presentado en Artificial Intelligence and Applications / 794: Modelling, Identification and Control / 795: Parallel and Distributed Computing and Networks / 796: Software Engineering / 792: Web-based Education, mar. 2013, doi: 10.2316/P.2013.796-012.

[131] D. Ståhl y J. Bosch, «Modeling continuous integration practice differences in industry software development», *J. Syst. Softw.*, vol. 87, pp. 48-59, ene. 2014, doi: 10.1016/j.jss.2013.08.032.

[132] A. Eck, F. Uebernickel, y W. Brenner, «Fit for Continuous Integration: How Organizations Assimilate an Agile Practice», presentado en 20th Americas Conference on Information Systems, AMCIS 2014, ago. 2014.

[133] E. Smith, «Continuous testing», presentado en 17th International Conference on Testing Computer Software, 2000.

[134] B. A. Kitchenham y S. L. Pfleeger, «Principles of Survey Research Part 2: Designing a Survey», *SIGSOFT Softw Eng Notes*, vol. 27, n.º 1, pp. 18-20, ene. 2002, doi: 10.1145/566493.566495.

[135] B. A. Kitchenham y S. L. Pfleeger, «Principles of Survey Research: Part 3: Constructing a Survey Instrument», *SIGSOFT Softw Eng Notes*, vol. 27, n.º 2, pp. 20-24, mar. 2002, doi: 10.1145/511152.511155.

[136] B. Kitchenham y S. L. Pfleeger, «Principles of Survey Research Part 4: Questionnaire Evaluation», *SIGSOFT Softw Eng Notes*, vol. 27, n.º 3, pp. 20-23, may 2002, doi: 10.1145/638574.638580.

[137] B. Kitchenham y S. L. Pfleeger, «Principles of Survey Research: Part 5: Populations and Samples», *SIGSOFT Softw Eng Notes*, vol. 27, n.º 5, pp. 17-20, sep. 2002, doi: 10.1145/571681.571686.

[138] B. Kitchenham y S. L. Pfleeger, «Principles of Survey Research Part 6: Data Analysis», *SIGSOFT Softw Eng Notes*, vol. 28, n.º 2, pp. 24-27, mar. 2003, doi: 10.1145/638750.638758.

[139] R. Koschke, «Software Visualization in Software Maintenance, Reverse Engineering, and Re-engineering: A Research Survey», *J. Softw. Maint.*, vol. 15, n.º 2, pp. 87-109, mar. 2003, doi: 10.1002/smr.270.

[140] T. C. Lethbridge, «A Survey of the Relevance of Computer Science and Software Engineering Education», en *Proceedings of the 11th Conference on Software Engineering Education and Training*, Washington, DC, USA, 1998, pp. 0056-, Accedido: abr. 05, 2019. [En línea]. Disponible en: http://dl.acm.org/citation.cfm?id=522339.794252.

[141] L. R. VIJAYASARATHY y D. TURK, «Agile software development: A survey of early adopters», *J. Inf. Technol. Manag.*, vol. 19, ene. 2008.

[142] V. Garousi y J. Zhi, «A survey of software testing practices in Canada», *J. Syst. Softw.*, vol. 86, n.º 5, pp. 1354-1376, may 2013, doi: 10.1016/j.jss.2012.12.051.

[143] K. Mao, L. Capra, M. Harman, y Y. Jia, «A survey of the use of crowdsourcing in software engineering», *J. Syst. Softw.*, vol. 126, pp. 57-84, abr. 2017, doi: 10.1016/j.jss.2016.09.015.

[144] Z. Guo, J. Zhao, y Y. Guan, «Research on Test Automated Framework for Reverse Generation of Use Cases», presentado en 8th International Conference on Management and Computer Science (ICMCS 2018), oct. 2018, doi: 10.2991/icmcs-18.2018.1.

[145] L. Bass, I. Weber, y L. Zhu, *DevOps: A Software Architect's Perspective*, 1 edition. New York: Addison-Wesley Professional, 2015.

[146] Mascheroni, Maximiliano Agustin, Orue Mascheroni, Carla Agustina, Lezcano Airaldi, Andrea, y Irrazábal, Emanuel, «Problems and Solutions for Continuous Testing at the Industry: A Survey», *Computación y Sistemas (In Press)*, 2021.

[147] Q. Luo, F. Hariri, L. Eloussi, y D. Marinov, «An empirical analysis of flaky tests», en *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering - FSE 2014*, Hong Kong, China, 2014, pp. 643-653, doi: 10.1145/2635868.2635920.

[148] G. Meszaros, S. M. Smith, y J. Andrea, «The Test Automation Manifesto», en *Extreme Programming and Agile Methods - XP/Agile Universe 2003*, Berlin, Heidelberg, 2003, pp. 73-81, doi: 10.1007/978-3-540-45122-8_9.

[149] J. P. Kotter, *Leading Change*, Edición: 1R. Boston, Mass: Harvard Business Review Press, 2012.

[150] M. L. Manns y L. Rising, *Fearless Change: Patterns for Introducing New Ideas*, Edición: 1. Place of publication not identified: Addison-Wesley Professional, 2015.

[151] P. Pureur y M. Erder, «Continuous architecture and continuous delivery», en *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*, Massachusetts, USA: Morgan Kaufmann (Elsevier), 2015, pp. 103-129.

[152] L. Brader, H. Hilliker, y A. C. Wills, *Testing for Continuous Delivery with Visual Studio 2012*, Edición: 1. Microsoft patterns & practices, 2013.

[153] M. Fowler, «TestCoverage», *martinfowler.com*. https://martinfowler.com/bliki/TestCoverage.html (accedido dic. 25, 2019).

[154] M. A. Mascheroni y E. Irrazábal, «Framework para la creación y ejecución de pruebas automatizadas sobre servicios REST», presentado en XXII Congreso Argentino de Ciencias de la Computación (CACIC 2016)., 2016, Accedido: jul. 04, 2018. [En línea]. Disponible en: http://hdl.handle.net/10915/56635.

[155] M. A. Mascheroni y E. Irrazábal, «A Design Pattern Approach for RESTful tests: A case study», en *Obras colectivas en ciencias de la computación*, 1.ª ed., vol. 1, pp. 257-268.

[156] M. Fowler, «PageObject», *martinfowler.com*. https://martinfowler.com/bliki/PageObject.html (accedido ene. 01, 2020).

[157] D. Chelimsky, D. Astels, B. Helmkamp, D. North, Z. Dennis, y A. Hellesoy, *The RSpec Book: Behaviour Driven Development with RSpec, Cucumber, and Friends*, Edición: 1. Lewisville, Tex: Pragmatic Bookshelf, 2010.

[158] «Mobile farm for software testing | Proceedings of the 20th International Conference on Human-Computer Interaction with Mobile Devices and Services Adjunct». https://dl.acm.org/doi/abs/10.1145/3236112.3236117 (accedido ene. 11, 2020).

[159] J. Itkonen, M. V. Mantyla, y C. Lassenius, «Defect Detection Efficiency: Test Case Based vs. Exploratory Testing», en *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*, sep. 2007, pp. 61-70, doi: 10.1109/ESEM.2007.56.

[160] R. Osherove, *The Art of Unit Testing: with Examples in .NET*, Edición: 1. Greenwich, CT: Manning Publications, 2009.

[161] T. Kaczanowski, *Practical Unit Testing with TestNG and Mockito*. Kraków: Tomasz Kaczanowski, 2012.

[162] G. M. Hall, «Unit Testing», en *Pro WPF and Silverlight MVVM: Effective Application Development with Model-View-ViewModel*, G. M. Hall, Ed. Berkeley, CA: Apress, 2010, pp. 145-162.

[163] G. Meszaros, *xUnit Test Patterns: Refactoring Test Code*, Edición: 1. Upper Saddle River, NJ: Addison-Wesley, 2007.

[164] N. B. Moe, D. Cruzes, T. Dybå, y E. Mikkelsen, «Continuous software testing in a globally distributed project», en *2015 IEEE 10th International Conference on Global Software Engineering*, jul. 2015, pp. 130-134, doi: 10.1109/ICGSE.2015.24.

[165] G. Gotimer y T. Stiehm, «DevOps Advantages for Testing: Increasing Quality through Continuous Delivery», *CrossTalk Magazine*, pp. 13-18, 2016.

[166] S. L. Joshi, B. Deshpande, y S. Punnekkat, «Experimental Analysis of Dependency Factors of Software Product Reliability using SonarQube», 2019.

[167] B. Barta, G. Manz, I. Siket, y R. Ferenc, «Challenges of SonarQube Plug-In Maintenance», en *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, feb. 2019, pp. 574-578, doi: 10.1109/SANER.2019.8667988.

[168] C. Vassallo, S. Panichella, F. Palomba, S. Proksch, A. Zaidman, y H. C. Gall, «Context is king: The developer perspective on the usage of static analysis tools», en *2018 IEEE 25th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, mar. 2018, pp. 38-49, doi: 10.1109/SANER.2018.8330195.

[169] C. Vassallo, F. Palomba, A. Bacchelli, y H. C. Gall, «Continuous code quality: are we (really) doing that?», en *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, Montpellier, France, sep. 2018, pp. 790-795, doi: 10.1145/3238147.3240729.

[170] D. Marcilio, R. Bonifácio, E. Monteiro, E. Canedo, W. Luz, y G. Pinto, «Are Static Analysis Violations Really Fixed? A Closer Look at Realistic Usage of SonarQube», en *2019 IEEE/ACM 27th International Conference on Program Comprehension (ICPC)*, may 2019, pp. 209-219, doi: 10.1109/ICPC.2019.00040.

[171] L. Eloussi, «Determining flaky tests from test failures», M.S. Thesis, University of Illinois at Urbana-Champaign, 2015.

[172] J. Penix, «Large-scale test automation in the cloud (Invited Industrial Talk)», 2012, p. 1122.

[173] M. Felderer, M. Büchler, M. Johns, A. Brucker, R. Breu, y A. Pretschner, «Security Testing: A Survey», 2016, pp. 1-51.

[174] M. Paul, *Official (ISC)2 Guide to the CSSLP ((ISC)2 Press)*, Edición: 1. Boca Raton, FL: CRC Press, 2011.

[175] D. Cruzes, M. Felderer, T. D. Oyetoyan, M. Gander, y I. Pekaric, «How is Security Testing Done in Agile Teams? A Cross-Case Analysis of Four Software Teams», may 2017, doi: 10.1007/978-3-319-57633-6_13.

[176] J. Engblom, «Virtual to the (near) end - Using virtual platforms for continuous integration», en *2015 52nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, jun. 2015, pp. 1-6, doi: 10.1145/2744769.2747948.

[177] S. Tilley y B. Floss, *Hard problems in software testing: Solutions using testing as a service*. Morgan & Claypool Publishers, 2014.

[178] M. Eyl, C. Reichmann, y K. Müller-Glaser, «Fast Feedback from Automated Tests Executed with the Product Build», en *Software Quality. The Future of Systems- and Software Development*, 2016, pp. 199-210.

[179] M. Sinisalo, «Improving Web User Interface Test Automation in Continuous Integration», M.S. Thesis, Tampere University of Technology, Finland, 2016.

[180] M. Virmani, «Understanding DevOps & Bridging the gap from continuous integration to continuous delivery», en *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, may 2015, pp. 78-82, doi: 10.1109/INTECH.2015.7173368.

[181] Colantonio, Joe, «How to Reuse Functional Test for Performance Testing», feb. 25, 2020. .

[182] D. C. Brabham, *Crowdsourcing*. Cambridge, Massachusetts ; London, England: The MIT Press, 2013.

[183] E. Dolstra, R. Vliegendhart, y J. Pouwelse, «Crowdsourcing GUI Tests», en *Verification and Validation 2013 IEEE Sixth International Conference on Software Testing*, mar. 2013, pp. 332-341, doi: 10.1109/ICST.2013.44.

[184] I. N. Sttrycek y M. L. Villán, «Desarrollo de una Plataforma de Crowdsourcing para realizar pruebas de software», Universidad Nacional del Nordeste, 2019.

[185] Members Of The Assessment Method Integrated Team, «Standard CMMI Appraisal Method for Process Improvement (SCAMPI), Version 1.1: Method Definition Document». https://resources.sei.cmu.edu/library/asset-view.cfm?assetid=5325 (accedido oct. 31, 2020).

[186] Garzás Parra, Javier, Irrazábal, Emanuel, y Santa Escolástica, Roberto, «Guía práctica de supervivencia en una auditoría CMMI», *Boletín de la ETSII*, Escuela Técnica Superior de Ingeniería Informática. Universidad Rey Juan Carlos, 2011.

[187] N. Baddoo y T. Hall, «Motivators of Software Process Improvement: an analysis of practitioners' views», *J. Syst. Softw.*, vol. 62, n.º 2, pp. 85-96, may 2002, doi: 10.1016/S0164-1212(01)00125-X.

[188] G. Coleman y R. O'Connor, «Investigating software process in practice: A grounded theory perspective», *J. Syst. Softw.*, vol. 81, n.º 5, pp. 772-784, may 2008, doi: 10.1016/j.jss.2007.07.027.

[189] E. Herranz, R. Colomo-Palacios, A. de Amescua, y M. Sánchez-Gordón, «Towards a gamification framework for software process improvement initiatives: Construction and validation», vol. 22, pp. 1509-1532, ene. 2016.

[190] B. Freimut, L. C. Briand, y F. Vollei, «Determining inspection cost-effectiveness by combining project data and expert opinion», *IEEE Trans. Softw. Eng.*, vol. 31, n.º 12, pp. 1074-1092, dic. 2005, doi: 10.1109/TSE.2005.136.

[191] Fehring, RJ, «The Fehring model», en *Classification of nursing diagnoses*, Philadelphia: Lippincott, 1994, pp. 55-62.

[192] I. D. C. Machado, J. D. Mcgregor, Y. C. Cavalcanti, y E. S. De Almeida, «On Strategies for Testing Software Product Lines: A Systematic Literature Review», *Inf Softw Technol*, vol. 56, n.º 10, pp. 1183-1199, oct. 2014, doi: 10.1016/j.infsof.2014.04.002.

[193] T. Dybå y T. Dingsøyr, «Strength of Evidence in Systematic Reviews in Software Engineering», en *Proceedings of the Second ACM-IEEE International Symposium on Empirical Software Engineering and Measurement*, New York, NY, USA, 2008, pp. 178-187, doi: 10.1145/1414004.1414034.

[194] J. Vilela, J. Castro, L. E. G. Martins, y T. Gorschek, «Integration between requirements engineering and safety analysis: A systematic literature review», *J. Syst. Softw.*, vol. 125, pp. 68-92, mar. 2017, doi: 10.1016/j.jss.2016.11.031.

[195] M. Alshraideh, «A Complete Automation of Unit Testing for JavaScript Programs», *J. Comput. Sci.*, vol. 4, dic. 2008, doi: 10.3844/jcssp.2008.1012.1019.

[196] E. Gomede, R. Thiago Da Silva, y R. Barros, «A Practical Approach to Software Continuous Delivery Focused on Application Lifecycle Management», jul. 2015, pp. 320-325, doi: 10.18293/SEKE2015-126.

[197] N. Li, A. Escalona, Y. Guo, y J. Offutt, «A Scalable Big Data Test Framework», en *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, abr. 2015, pp. 1-2, doi: 10.1109/ICST.2015.7102619.

[198] P. Zhang, X. Zhou, W. Li, y J. Gao, «A Survey on Quality Assurance Techniques for Big Data Applications», en *2017 IEEE Third International Conference on Big Data Computing Service and Applications (BigDataService)*, abr. 2017, pp. 313-319, doi: 10.1109/BigDataService.2017.42.

[199] W. T. Tsai, P. Zhong, J. Balasooriya, Y. Chen, X. Bai, y J. Elston, «An approach for service composition and testing for cloud computing», en *2011 Tenth International Symposium on Autonomous Decentralized Systems*, mar. 2011, pp. 631-636, doi: 10.1109/ISADS.2011.90.

[200] E. Smith, «Automated Test Results Processing», presentado en STAREAST 2001 - Software Testing Conference, 2001.

[201] J. Gmeiner, R. Ramler, y J. Haslinger, «Automated testing in the continuous delivery pipeline: A case study of an online company», en *2015 IEEE Eighth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, abr. 2015, pp. 1-6, doi: 10.1109/ICSTW.2015.7107423.

[202] J. Campos, G. Fraser, A. Arcuri, y R. Abreu, «Continuous Test Generation on Guava», sep. 2015, pp. 228-234, doi: 10.1007/978-3-319-22183-0_16.

[203] M. Kawalerowicz y L. Madeyski, «Continuous Test-Driven Development - A Novel Agile Software Development Practice and Supporting Tool», presentado en ENASE 2013 - Proceedings of the 8th International Conference on Evaluation of Novel Approaches to Software Engineering, jul. 2013, doi: 10.5220/0004587202600267.

[204] D. Saff y M. D. Ernst, «Continuous Testing in Eclipse», *Electron. Notes Theor. Comput. Sci.*, vol. 107, pp. 103-117, dic. 2004, doi: 10.1016/j.entcs.2004.02.051.

[205] M. Zachow, «Designing an Android Continuous Delivery pipeline», 2016.

[206] D. Marijan y M. Liaaen, «Effect of Time Window on the Performance of Continuous Regression Testing», en *2016 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, oct. 2016, pp. 568-571, doi: 10.1109/ICSME.2016.77.

[207] J. Kim, H. Jeong, y E. Lee, «Failure History Data-based Test Case Prioritization for Effective Regression Test», en *Proceedings of the Symposium on Applied Computing*, New York, NY, USA, 2017, pp. 1409-1415, doi: 10.1145/3019612.3019831.

[208] E. Alégroth y R. Feldt, «Industrial Application of Visual GUI Testing: Lessons Learned», en *Continuous Software Engineering*, J. Bosch, Ed. Cham: Springer International Publishing, 2014, pp. 127-140.

[209] M. Burgin y N. C. Debnath, «Intelligent testing systems», *World Autom. Congr.*, pp. 1-6, 2010.

[210] B. Busjaeger y T. Xie, «Learning for Test Prioritization: An Industrial Case Study», en *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2016, pp. 975-980, doi: 10.1145/2950290.2983954.

[211] D. I. Savchenko, G. I. Radchenko, y O. Taipale, «Microservices validation: Mjolnirr platform case study», en *2015 38th International Convention on Information and Communication Technology, Electronics and Microelectronics (MIPRO)*, may 2015, pp. 235-240, doi: 10.1109/MIPRO.2015.7160271.

[212] G. Brajnik, A. Baruzzo, y S. Fabbro, «Model-Based Continuous Integration Testing of Responsiveness of Web Applications», en *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, abr. 2015, pp. 1-2, doi: 10.1109/ICST.2015.7102626.

[213] D. Marijan, «Multi-perspective Regression Test Prioritization for Time-Constrained Environments», en *2015 IEEE International Conference on Software Quality, Reliability and Security*, ago. 2015, pp. 157-162, doi: 10.1109/QRS.2015.31.

[214] A. A Alotaibi y M. R. Qureshi, «Novel Framework for Automation Testing of Mobile Applications using Appium», *Int. J. Mod. Educ. Comput. Sci. IJMECS*, vol. 9, pp. 34-40, feb. 2017, doi: 10.5815/ijmecs.2017.02.04.

[215] A. Gambi, A. Gorla, y A. Zeller, «O!Snap: Cost-Efficient Testing in the Cloud», en *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, mar. 2017, pp. 454-459, doi: 10.1109/ICST.2017.51.

[216] E. Alégroth y R. Feldt, «On the long-term use of visual gui testing in industrial practice: a case study», *Empir. Softw. Eng.*, vol. 22, n.º 6, pp. 2937-2971, dic. 2017, doi: 10.1007/s10664-016-9497-6.

[217] A. Ramakrishnan y R. Manjula, «Perceptual Difference for Safer Continuous Delivery», *Int. Res. J. Eng. Technol. IRJET*, vol. 3, n.º 11, pp. 793-798, 2016.

[218] P. Pureur y M. Erder, «Principles of Continuous Architecture», en *Continuous Architecture: Sustainable Architecture in an Agile and Cloud-Centric World*, Massachusetts, USA: Morgan Kaufmann (Elsevier), 2015, pp. 21-37.

[219] H. Hemmati, Z. Fang, y M. V. Mantyla, «Prioritizing Manual Test Cases in Traditional and Rapid Release Environments», en *2015 IEEE 8th International Conference on Software Testing, Verification and Validation (ICST)*, abr. 2015, pp. 1-10, doi: 10.1109/ICST.2015.7102602.

[220] D. Saff y M. D. Ernst, «Reducing wasted development time via continuous testing», en *14th International Symposium on Software Reliability Engineering, 2003. ISSRE 2003.*, nov. 2003, pp. 281-292, doi: 10.1109/ISSRE.2003.1251050.

[221] A. Cavalli, S. Maag, y G. Morales, «Regression and Performance Testing of an e-Learning Web Application: dotLRN», en *2007 Third International IEEE Conference on Signal-Image Technologies and Internet-Based System*, dic. 2007, pp. 369-376, doi: 10.1109/SITIS.2007.129.

[222] E. Knauss, M. Staron, W. Meding, O. Söder, A. Nilsson, y M. Castell, «Supporting Continuous Integration by Code-Churn Based Test Selection», en *2015 IEEE/ACM 2nd International Workshop on Rapid Continuous Software Engineering*, may 2015, pp. 19-25, doi: 10.1109/RCoSE.2015.11.

[223] S. Elbaum, G. Rothermel, y J. Penix, «Techniques for Improving Regression Testing in Continuous Integration Development Environments», en *Proceedings of the 22Nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*, New York, NY, USA, 2014, pp. 235-245, doi: 10.1145/2635868.2635910.

[224] D. Marijan, A. Gotlieb, y S. Sen, «Test Case Prioritization for Continuous Regression Testing: An Industrial Case Study», en *2013 IEEE International Conference on Software Maintenance*, sep. 2013, pp. 540-543, doi: 10.1109/ICSM.2013.91.

[225] N. Rathod y A. Surve, «Test orchestration: a framework for Continuous Integration and Continuous deployment», en *2015 International Conference on Pervasive Computing (ICPC)*, ene. 2015, pp. 1-5, doi: 10.1109/PERVASIVE.2015.7087120.

[226] D. Marijan y M. Liaaen, «Test Prioritization with Optimally Balanced Configuration Coverage», en *2017 IEEE 18th International Symposium on High Assurance Systems Engineering (HASE)*, ene. 2017, pp. 100-103, doi: 10.1109/HASE.2017.26.

[227] S. W. Ambler, «Test-Driven Development of Relational Databases», *IEEE Softw*, vol. 24, n.º 3, pp. 37-43, may 2007, doi: 10.1109/MS.2007.91.

[228] Z. Zhang, Z. Tong, y X. Gao, «Testing in parallel: A need for practical regression testing», presentado en 5th International Conference on Software and Data Technologies (ICSOFT), 2010.

[229] J. Micco, *The State of Continuous Integration Testing @Google*. 2017.

[230] D. Marijan, M. Liaaen, A. Gotlieb, S. Sen, y C. Ieva, «TITAN: Test Suite Optimization for Highly Configurable Software», en *2017 IEEE International Conference on Software Testing, Verification and Validation (ICST)*, mar. 2017, pp. 524-531, doi: 10.1109/ICST.2017.60.

[231] W. M. Watanabe, R. P. M. Fortes, y A. L. Dias, «Using Acceptance Tests to Validate Accessibility Requirements in RIA», en *Proceedings of the International Cross-Disciplinary Conference on Web Accessibility*, New York, NY, USA, 2012, p. 15:1-15:10, doi: 10.1145/2207016.2207022.

[232] M. Fagerström *et al.*, «Verdict machinery: On the need to automatically make sense of test results», en *Proceedings of the 25th International Symposium on Software Testing and Analysis*, New York, NY, USA, 2016, pp. 225-234, doi: 10.1145/2931037.2931064.

[233] D. Dermeval *et al.*, «Applications of ontologies in requirements engineering: a systematic review of the literature», *Requir. Eng.*, vol. 21, n.º 4, pp. 405-437, nov. 2016, doi: 10.1007/s00766-015-0222-6.

[234] S. Easterbrook, J. Singer, M.-A. Storey, y D. Damian, «Selecting Empirical Methods for Software Engineering Research», en *Guide to Advanced Empirical Software Engineering*, F. Shull, J. Singer, y D. I. K. Sjøberg, Eds. London: Springer London, 2008, pp. 285-311.

[235] R. Wieringa, N. Maiden, N. Mead, y C. Rolland, «Requirements Engineering Paper Classification and Evaluation Criteria: A Proposal and a Discussion», *Requir Eng*, vol. 11, n.º 1, pp. 102-107, dic. 2005, doi: 10.1007/s00766-005-0021-6.

[236] L.-R. Yang, C.-F. Huang, y K.-S. Wu, «The association among project manager's leadership style, teamwork and project success», *Int. J. Proj. Manag.*, vol. 29, n.º 3, pp. 258-267, abr. 2011, doi: 10.1016/j.ijproman.2010.03.006.

[237] J. T. O'Connor y L.-R. Yang, «Project Performance versus Use of Technologies at Project and Phase Levels», *J. Constr. Eng. Manag.-Asce - J CONSTR ENG MANAGE-ASCE*, vol. 130, jun. 2004, doi: 10.1061/(ASCE)0733-9364(2004)130:3(322).

[238] J. Dougherty, R. Kohavi, y M. Sahami, «Supervised and Unsupervised Discretization of Continuous Features», en *Machine Learning Proceedings 1995*, A. Prieditis y S. Russell, Eds. San Francisco (CA): Morgan Kaufmann, 1995, pp. 194-202.

[239] J. M. Carrillo de Gea, J. Nicolás, J. L. Fernández Alemán, A. Toval, C. Ebert, y A. Vizcaíno, «Requirements engineering tools: Capabilities, survey and assessment», *Inf. Softw. Technol.*, vol. 54, n.º 10, pp. 1142-1157, oct. 2012, doi: 10.1016/j.infsof.2012.04.005.