

Universidad Nacional de Misiones

Facultad de Ciencias Exactas, Químicas y Naturales

**Trabajo Final de Maestría en Tecnologías de la
Información**

**Diseño y construcción de una metodología de
pruebas continuas aplicable al modelo de
desarrollo de software dentro del
Sistema de Información Universitaria**

Autor: Lic. Sergio Fabián Vier

Director: Dr. Emanuel Irrazabal

Año 2021

...especialmente dedicado a Carolina, a mis padres, a mis amigos.

Resumen

En este trabajo se presenta una metodología para realizar pruebas continuas, aplicable al modelo de desarrollo de software que se lleva a cabo en el Sistema de Información Universitaria. La metodología se diseña y construye aplicando métodos, procesos y herramientas que son factibles de adoptar en los proyectos llevados a cabo por el Sistema de Información Universitaria.

El desarrollo continuo de software es un enfoque que busca entregar software funcional en ciclos reducidos. Involucra actividades como construir y garantizar que el producto mantenga un cierto nivel de calidad, todo en un ciclo continuo o repetitivo. La automatización de estas actividades ayuda a establecer mecanismos que complementan el soporte a este enfoque, en particular para las pruebas de software.

El marco de trabajo propuesto en este trabajo se valida mediante su adopción en el ámbito de desarrollo del proyecto SIU-Araí. Esta implementación permitió la exploración de soluciones tecnológicas que se integran a las necesidades del Sistema de Información Universitaria.

Palabras Claves: Pruebas continuas, Integración continua, entrega continua, desarrollo continuo, conducto de despliegue, SIU

Abstract

In this thesis a methodology is presented for continuous testing, applicable to the software development model that is in use at the University Information System. The methodology is designed and built by applying methods, processes and tools that are feasible to adopt in the projects carried out by the University Information System.

Continuous software development is an approach that seeks to deliver functional software in short cycles. It involves activities such as building and ensuring that the product maintains a certain level of quality, all in a continuous or repetitive cycle. Automating these activities helps establish mechanisms that complement support for this approach, particularly for software testing.

The framework proposed in this work is validated by its adoption in the development of SIU-Araí project. This implementation allowed the exploration of technological solutions that are integrated to the needs of the University Information System.

Keywords: Continuous testing, continuous integration, continuous delivery, continuous development, pipeline, SIU

Reconocimientos

Quiero agradecer a Carolina, mi compañera de vida, que con su cariño y apoyo logró mantenerme en el camino y así alcanzar mis objetivos.

A mis padres, que me enseñaron a no dejar de buscar la perfección en lo que hacemos a diario.

Al Sistema de Información Universitaria, que recibió con los brazos abiertos este proyecto, en especial a Andrés Blanco con su apoyo continuo para impulsar el cambio.

A la Universidad Nacional de Misiones, que me brindó la oportunidad de lograr una formación humana de calidad.

Finalmente, al Dr. Emanuel Irrazabal, por convencerme de que una tarea titánica no significa que sea imposible de concretar.

Índice General

1	INTRODUCCIÓN	14
1.1	Fundamentación de la Problemática	14
1.2	Hipótesis y Objetivos	16
1.2.1	Objetivo General.....	16
1.2.2	Objetivos Específicos	16
1.3	Áreas de Conocimiento Disciplinar	17
1.4	Marco de Investigación	18
1.5	Método de Investigación	19
1.5.1	Etapas de Documentación	20
1.6	Estructura del Trabajo	22
2	MARCO TEÓRICO	26
2.1	Desarrollo Continuo de Software	26
2.1.1	Integración Continua	26
2.1.1.1	El proceso de integración continua.....	28
2.1.1.2	Componentes de la Integración Continua	29
2.1.1.3	Flujo de Integración Continua.....	31
2.1.2	Despliegue Continuo	32
2.1.2.1	Despliegue tradicional y despliegue continuo.....	33
2.1.2.2	Etapas del despliegue continuo.....	35
2.1.3	Entrega Continua.....	37
2.1.3.1	Entrega Continua y Despliegue Continuo.....	37
2.1.3.2	Componentes de entrega continua.....	39
2.1.3.3	Conducto de despliegue	41
2.1.3.4	Lanzamientos rápidos	43
2.1.4	Pruebas Continuas	43
2.1.4.1	Análisis estático de código fuente	44
2.1.4.2	Pruebas Unitarias	46
2.1.4.3	Pruebas de Integración	47
2.1.4.4	Pruebas Funcionales.....	48
2.1.4.5	Pruebas No funcionales	49
2.1.4.6	Pruebas de aceptación del usuario	50

2.1.5	Las Pruebas en el Desarrollo Continuo de Software	52
2.2	Revisión Sistemática de la Literatura.....	56
2.2.1	Introducción	56
2.2.2	Método de Investigación.....	57
2.2.2.1	Objetivo y Preguntas de Investigación	58
2.2.2.2	El Proceso de Búsqueda	58
2.2.2.3	Selección de los Estudios.....	59
2.2.2.4	Proceso de Extracción de los Datos.....	59
2.2.2.5	Evaluación de Calidad.....	61
2.2.3	Resultados	62
2.2.4	Conclusiones.....	69
2.3	Pruebas de Software en el marco del SIU.....	70
2.3.1	Breve historia	70
2.3.1.1	Objetivos y Actividades	72
2.3.1.2	Organización	73
2.3.1.2.1	Modalidad de trabajo colaborativo.....	74
2.3.1.2.2	Gestión orientada a proyectos.....	75
2.3.2	Cómo desarrolla el Software	77
2.3.2.1	Conformación de los equipos de trabajo	77
2.3.2.2	Proceso para Desarrollo de Software.....	79
2.3.2.2.1	Flujos de apoyo al proceso de desarrollo de software.....	82
2.3.2.2.2	Publicación de nuevas versiones de los proyectos.....	83
2.3.3	Cómo realiza las Pruebas de Software	86
2.3.3.1	Pruebas unitarias.....	86
2.3.3.2	Pruebas de interfaz de usuario.....	86
2.3.3.3	Pruebas de API	87
2.3.3.4	Pruebas de instalación	88
2.3.3.5	Pruebas de integración.....	89
2.3.3.6	Pruebas de aceptación.....	89
2.3.3.7	Pruebas de regresión	89
2.3.3.8	Pruebas de compatibilidad	90
2.3.3.9	Pruebas de rendimiento	90
2.3.4	Conclusiones.....	91
3	DESARROLLO	94

3.1	Las Prácticas Continuas y las Pruebas de Software	94
3.2	Desarrollo de Software en el SIU.....	96
3.2.1	Resumen de la situación actual.....	96
3.2.2	Planteo de la solución	97
3.2.3	Definición del conducto de despliegue	98
3.2.3.1	Personalización del conducto de despliegue.....	99
3.2.3.2	Modelado en lenguaje SPEM	100
3.3	Modelo de procesos para pruebas continuas en el SIU	101
3.3.1	Proceso para construcción de la aplicación	102
3.3.1.1	Etapa A1.1: construcción.....	103
3.3.1.2	Etapa A1.2: pruebas.....	105
3.3.1.3	Etapa A1.3: calidad	107
3.3.1.4	Etapa A1.4: despliegue	109
3.3.1.5	Etapa A1.5: pruebas despliegue.....	111
3.3.1.6	Etapa A1.6: publicación	114
3.3.2	Proceso para distribución de la aplicación con Docker	116
3.3.2.1	Etapa B1.1: construcción.....	117
3.3.2.2	Etapa B1.2: pruebas.....	119
3.3.2.3	Etapa B1.3: calidad	121
3.3.2.4	Etapa B1.4: despliegue	123
3.3.2.5	Etapa B1.5: pruebas despliegue.....	125
3.3.2.6	Etapa B1.6: publicación	128
3.3.3	Etapas del proceso según el flujo de desarrollo	129
3.3.3.1	Proceso para la construcción de la aplicación.....	130
3.3.3.2	Proceso para la distribución de la aplicación con Docker.....	130
4	VERIFICACIÓN	132
4.1	Definición del ecosistema software.....	132
4.2	Preparación del proyecto SIU-Araí	134
4.2.1	Arquitectura	134
4.2.2	Gestión de la configuración	135
4.2.3	Empaquetado	135
4.3	Implementación del Proceso de Pruebas Continuas	136
4.3.1	Selección de procesos del modelo	136
4.3.2	Conducto de despliegue.....	137

4.3.2.1	Etapa 1: Construcción	138
4.3.2.2	Etapa 2: Pruebas.....	140
4.3.2.3	Etapa 3: Calidad	140
4.3.2.4	Etapa 4: Despliegue	142
4.3.2.5	Etapa 5: Pruebas despliegue.....	143
4.3.2.6	Etapa 6: Publicación.....	144
4.4	Resultados y Discusiones.....	144
4.4.1	Pruebas que fueron automatizadas.....	144
4.4.2	Retroalimentación.....	145
4.4.3	Estadísticas	149
4.4.4	Discusión de los resultados.....	152
5	CONCLUSIONES	158
5.1	Análisis del cumplimiento de los Objetivos	158
5.2	Conclusiones generales	160
5.3	Trabajo Futuro	161
5.3.1	Expandir el ámbito de aplicación de las pruebas continuas	161
5.3.2	Mejora de la evaluación de las pruebas en desarrollo	161
5.3.3	Optimizar la ejecución del Pipeline.....	162
6	REFERENCIAS	164
7	ANEXOS.....	174
7.1	Docker para empaquetar aplicaciones	174
7.1.1	Contenedor de una aplicación.....	174
7.1.2	Orquestación de contenedores	175
7.1.3	Construcción de imagen Docker	176
7.2	Gitlab: plataforma para CI/CD.....	181
7.2.1	Flujo de trabajo.....	182
7.2.2	Merge Requests	183
7.2.3	Pipeline.....	184

Índice de Figuras

<i>Figura 1 SWEBOK, áreas de conocimiento básicas</i>	<i>17</i>
<i>Figura 2 Articulación de proyectos y equipos de trabajo en el SIU</i>	<i>19</i>
<i>Figura 3 Pasos del método de investigación científica.....</i>	<i>20</i>
<i>Figura 4 Fases del proceso de revisión sistemática de la literatura</i>	<i>21</i>
<i>Figura 5 Escenario de integración continua y sus componentes.....</i>	<i>28</i>
<i>Figura 6 Flujo de actividades en la integración continua.....</i>	<i>31</i>
<i>Figura 7 Influencia de los principios de desarrollo de software Ágil y Lean</i>	<i>32</i>
<i>Figura 8 Diferencias en el pase a producción entre CDE y CD.....</i>	<i>38</i>
<i>Figura 9 Conducto de despliegue, DP, propuesto por Humble y Farley.....</i>	<i>41</i>
<i>Figura 10 Herramientas típicas utilizadas para la implementación de un DP.....</i>	<i>42</i>
<i>Figura 11 Ciclo de desarrollo de software tradicional.....</i>	<i>53</i>
<i>Figura 12 Costo de solucionar defectos en el software.....</i>	<i>53</i>
<i>Figura 13 Etapas en un proceso de desarrollo de software iterativo.....</i>	<i>54</i>
<i>Figura 14 Proceso de búsqueda y selección de estudios</i>	<i>60</i>
<i>Figura 15 Distribución de las publicaciones a lo largo de los años</i>	<i>63</i>
<i>Figura 16 Tópicos de investigación analizados en torno a CI</i>	<i>64</i>
<i>Figura 17 Métodos relacionados con las prácticas continuas</i>	<i>65</i>
<i>Figura 18 Dificultades relacionadas con la integración continua.....</i>	<i>66</i>
<i>Figura 19 Herramientas utilizadas para dar soporte a las prácticas continuas.....</i>	<i>67</i>
<i>Figura 20 Tipos y tamaños de aplicaciones reportados en los estudios</i>	<i>68</i>
<i>Figura 21 Tipos de lenguajes de programación</i>	<i>69</i>
<i>Figura 22 Resumen de la ejecución de proyectos SIU, en números.....</i>	<i>71</i>
<i>Figura 23 Indicadores de la ejecución del SIU</i>	<i>72</i>
<i>Figura 24 Flujo de información entre UUNN y organismos</i>	<i>76</i>
<i>Figura 25 Tipos de proyectos SIU de acuerdo a la pirámide del conocimiento</i>	<i>77</i>
<i>Figura 26 Agrupación de roles por perfil para equipos de trabajo del SIU</i>	<i>78</i>
<i>Figura 27 Metodología de desarrollo de software del SIU.....</i>	<i>80</i>
<i>Figura 28 Estados de los tickets en el sistema de Gestión de Solicitudes</i>	<i>82</i>
<i>Figura 29 Transición de estados en la gestión del proyecto SIU-Diaguita</i>	<i>83</i>
<i>Figura 30 Publicación de versiones, por proyecto.....</i>	<i>85</i>
<i>Figura 31 Tiempo entre la publicación de cada versión, por proyecto</i>	<i>85</i>
<i>Figura 32 DP propuesto para el desarrollo de software en el SIU.....</i>	<i>99</i>
<i>Figura 33 Tablero de Pipelines, con resultados de los últimos ejecutados</i>	<i>146</i>

<i>Figura 34 Visualización de los resultados de las tareas de un pipeline.....</i>	<i>147</i>
<i>Figura 35 Visualización de los resultados de un pipeline de tareas reducido.....</i>	<i>148</i>
<i>Figura 36 Visualización de un MR en Gitlab, con información del pipeline</i>	<i>149</i>
<i>Figura 37 Cantidad de pipelines ejecutados.....</i>	<i>150</i>
<i>Figura 38 Duración de los últimos 30 pipelines (a 25 de abril de 2020).....</i>	<i>150</i>
<i>Figura 39 Pipelines ejecutados en el periodo del mes (al 25 de abril de 2020)..</i>	<i>151</i>
<i>Figura 40 Pipelines ejecutados en el periodo del año (al 25 de abril de 2020) ..</i>	<i>152</i>
<i>Figura 41 Contraste del tiempo requerido para el despliegue</i>	<i>155</i>
<i>Figura 42 Distribución de errores detectados a lo largo del desarrollo.....</i>	<i>155</i>
<i>Figura 43 Acciones típicas en un pipeline de Gitlab.....</i>	<i>184</i>

Índice de Tablas

<i>Tabla 1: Diferencias entre el despliegue tradicional y el continuo</i>	<i>34</i>
<i>Tabla 2: Diferencias y similitudes entre CDE y CD</i>	<i>38</i>
<i>Tabla 3: Escala de evaluación de los criterios DARE.....</i>	<i>61</i>
<i>Tabla 4: Listado de estudios secundarios o RSL obtenidos para su análisis</i>	<i>62</i>
<i>Tabla 5: Datos de análisis sobre la publicación de versiones de los proyectos ...</i>	<i>84</i>
<i>Tabla 6: Uso de las etapas del proceso para construcción de la aplicación</i>	<i>130</i>
<i>Tabla 7: Uso de las etapas del proceso para distribución de la aplicación</i>	<i>130</i>
<i>Tabla 8: Selección de tareas según la fase del DP, implantadas en el proyecto SIU-Arai: Usuarios.....</i>	<i>136</i>
<i>Tabla 9: Tareas a ejecutar en las diferentes ramas de desarrollo del proyecto SIU-Arai: Usuarios.....</i>	<i>137</i>

1

INTRODUCCIÓN

1 INTRODUCCIÓN

En este primer capítulo se introducen los motivos que impulsaron la definición de este TFM, junto con la problemática actual y sus fundamentos. Se detallan los objetivos planteados que guiarán el resto del trabajo. Finalmente, se describe el método y marco de investigación utilizados para desarrollar este trabajo.

1.1 Fundamentación de la Problemática

El Sistema de Información Universitaria (SIU) [1], [2] es una institución que inició sus pasos en el año 1996 desarrollando sistemas informáticos para la gestión de diversas áreas en instituciones que componen el *Sistema Universitario Argentino*. El objetivo principal del SIU es la construcción de sistemas de gestión capaces de mejorar la calidad y disponibilidad de los datos y fortalecer la gestión de las instituciones universitarias, en cada uno de sus ámbitos de acción [3]. El SIU cuenta con 53 proyectos de software activos contabilizados al año 2018 [4], todos de diversos tamaños y en constante actividad como, por ejemplo, nuevas versiones, corrección de errores, soporte o mantenimiento a la instalación y actualización. Uno de los proyectos recientes y en activo desarrollo es el SIU-Araí [5]. Se trata de una plataforma integradora de servicios, donde cada uno de los otros sistemas desarrollados por el SIU es considerado un módulo dentro del Araí. Proporciona además servicios adicionales a los módulos que interactúen con ella, como ser acceso a la identidad de usuarios en forma centralizada. Esta plataforma está construida sobre una arquitectura orientada a servicios (del inglés Service Object Architecture, SOA) con la cual se soporta el concepto de integración de módulos, objetivos de la mejora de la gestión y calidad de los datos. De esta forma, la plataforma SIU-Araí es concebida como un componente central en la integración de las soluciones que proporciona el SIU hacia las instituciones que componen el *Sistema Universitario Argentino*.

Es clave entender que la industria del software se rige por paradigmas en los cuales la funcionalidad del software está en continua evolución [6]. De la misma manera, el SIU lleva a cabo el desarrollo de las soluciones software que

brinda haciendo uso de un enfoque evolutivo: agregando nuevas funcionalidades e innovaciones, mejoras y correcciones de forma continua; en definitiva buscando llegar a sus clientes lo antes posible y evolucionar el producto software de forma iterativa e incremental. Esto es cubierto por el enfoque de la ingeniería del software conocido como Desarrollo Continuo de Software (del inglés Continuous Software Development, CSD) [7]. Su aplicación redundante en beneficios, pero también acarrea nuevos desafíos [8]. El CSD está enfocado en producir software en ciclos de tiempo reducidos, con su respectivo lanzamiento, manteniéndose así en un ciclo continuo. Las Pruebas Continuas (del inglés Continuous Testing, CT), en conjunto con la Integración Continua (del inglés Continuous Integration, CI) y el Despliegue Continuo (del inglés Continuous Deployment, CD) forman parte de las técnicas que surgieron en este enfoque [9], [10] para dar soporte a las actividades que significan producir software entregable, útil y verificado, en un tiempo razonable. Una de las premisas del CSD es lograr la entrega de valor al usuario final: debe ser en tiempos aceptables pero también con un nivel de calidad y confiabilidad determinado, manejando los riesgos inherentes que pueden presentarse [8], [11], [12].

La necesidad de establecer una metodología de pruebas continuas para llevar a cabo actividades en el SIU se sustenta en que éstas aportan grandes beneficios al momento de reducir el riesgo asociado al desarrollo de las características del software [8], [13]. Entre ellas se puede mencionar que aumenta la confiabilidad en el servicio, permite la reducción del tiempo de salida al mercado así como brindar retroalimentación rápida y poner el foco en mejorar la calidad del producto en desarrollo, entre otros [13]. Además, cuanto antes un defecto sea capturado y solucionado, menor es el costo implicado en su solución [14]. Incidentalmente, debido al esfuerzo considerable para configurar un entorno para diversos tipos de pruebas, éstas no se ejecutaban durante etapas tempranas del desarrollo. Era factible ejecutarlo solamente al final de un gran ciclo de desarrollo. Con la llegada del CSD, la necesidad de ejecutar las pruebas en todo momento y durante todo ciclo de desarrollo pasa a ser primordial [9], [15].

El proyecto SIU-Araí por sus características, reciente trayectoria, innovación tecnológica, requerimientos en constante cambio y crecimiento, posee los principios del manifiesto ágil [16] y por ende los fundamentos de la entrega continua [15] son aplicables. Esto hace posible realizar actividades tales como la

gestión de la configuración, la automatización de las pruebas o la integración y despliegue continuo. En vista de estas condiciones, este proyecto es un caso de estudio adecuado para llevar a cabo la validación de este trabajo de maestría. De forma adicional, en la vinculación necesaria el Sistema de Información Universitaria se transforma en partícipe del proceso de análisis, evaluación y recepción de la experiencia tecnológica adquirida.

1.2 Hipótesis y Objetivos

La hipótesis planteada en este Trabajo Final de Maestría es la siguiente: *“es factible elaborar una metodología aplicable al desarrollo de software en el SIU, que permita la construcción de un proceso de pruebas continuas, así como la propuesta de un marco tecnológico para implementar y dar soporte a dicho proceso de pruebas continuas”.*

1.2.1 Objetivo General

Elaboración de una metodología aplicable al desarrollo de software en el SIU, que permita la construcción de un proceso de pruebas continuas como parte integral a un proceso de desarrollo continuo de software. Además, proponer un marco tecnológico basado en un ecosistema de herramientas adaptadas para implementar y dar soporte a dicho proceso de pruebas continuas.

1.2.2 Objetivos Específicos

1. Revisión de la literatura referente al estado de situación de las técnicas de pruebas continuas en el marco de la tecnología con la que se construye la plataforma SIU-Araí.
2. Análisis y evaluación del proceso actual de desarrollo de software dentro del Sistema de Información Universitaria.
3. Diseño de los procedimientos para construir una metodología personalizada de pruebas continuas, aplicables al ámbito específico del Sistema de Información Universitaria.

4. Definición y configuración de un ecosistema software que permita dar soporte a dicho conjunto de procedimientos, en un marco de desarrollo de software moderno y adaptable.
5. Validación de los procedimientos propuestos con la configuración del ecosistema software, utilizando como caso de estudio el desarrollo de la plataforma SIU-Araí.

1.3 Áreas de Conocimiento Disciplinar

Dentro del área de la Ingeniería del Software, y tomando como marco de referencia la guía Cuerpo del Conocimiento de la Ingeniería del Software (del inglés Software Engineering Body of Knowledge, SWEBOK) [17], este trabajo se enmarca en las áreas de conocimiento básicas referenciadas en la Figura 1 y descritas a continuación:

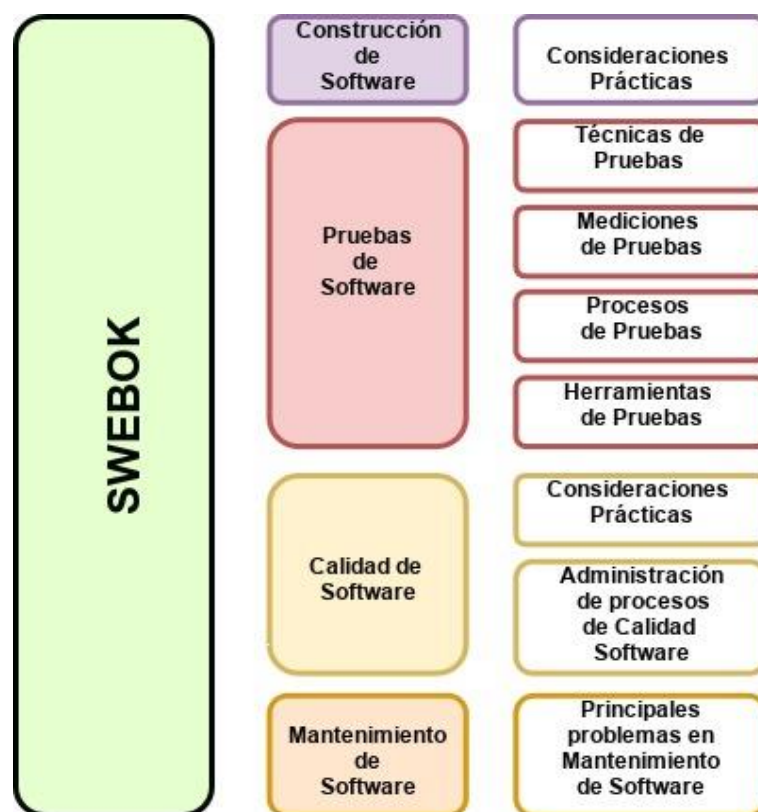


Figura 1 SWEBOK, áreas de conocimiento básicas

- Construcción de Software, con la subárea Consideraciones Prácticas (Practical Considerations) y en particular la sección Construcción de Pruebas (Construction Testing).
- Pruebas de Software, con las subáreas Técnicas de Pruebas (Test Techniques), Mediciones de Pruebas (Test Measures), Procesos de Prueba (Test Process) y Herramientas de Prueba de Software (Software Testing Tools).
- Calidad de Software, con las subáreas Consideraciones Prácticas (Practicals Considerations) y Administración de Procesos de Calidad de Software (Software Quality Management Process) en particular la sección Verificación y Validación (Verification & Validation).
- Mantenimiento de Software, con las subáreas Principales Problemas en Mantenimiento de Software (Key Issues in Software Maintenance) y en particular la sección de Pruebas (Testing).

1.4 Marco de Investigación

El trabajo será desarrollado dentro del ámbito del Sistema de Información Universitaria, en el área de desarrollo de software. Se trata de impulsar este trabajo final de maestría de tal forma que el resultado de la misma sea aplicable de forma general a los proyectos activos en el desarrollo y mantenimiento de software.

En la Figura 2 se expone el esquema de articulación entre los proyectos que maneja el SIU (tanto los orientados a la comunidad en general como los proyectos internos, con objetivos de integración y soporte hacia los primeros).

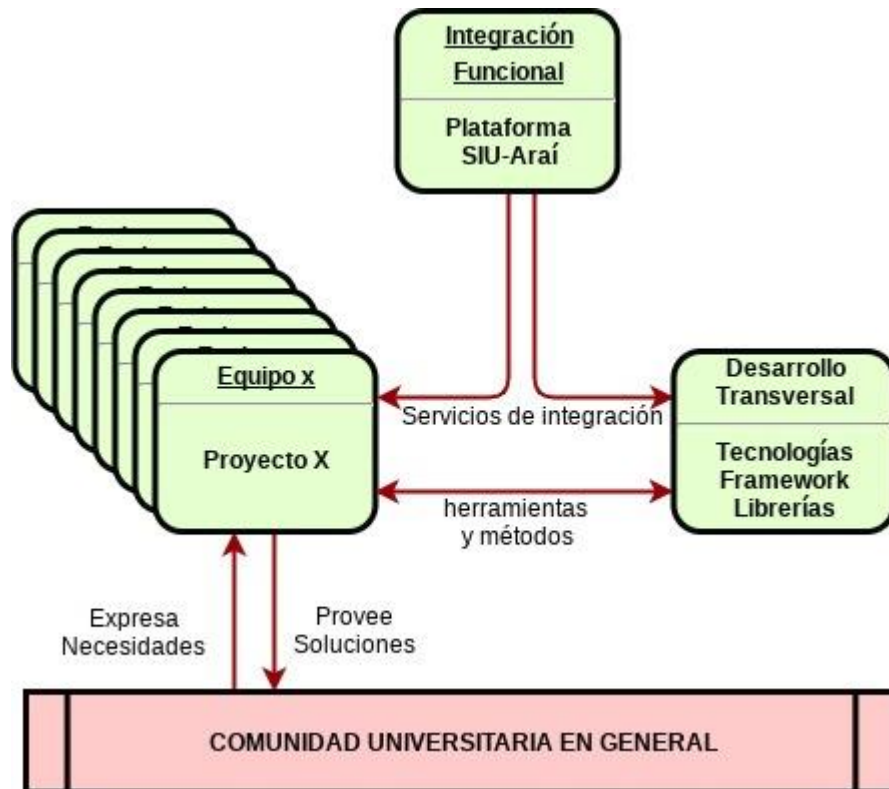


Figura 2 Articulación de proyectos y equipos de trabajo en el SIU

En primer lugar, existe un equipo que está dedicado al desarrollo de tecnologías, herramientas y métodos que son transversales o de apoyo a los proyectos generales. Aquí es donde se acopla el equipo denominado Integración Funcional, bajo el cual se ejecuta el proyecto Plataforma SIU-Araí. Este será el proyecto objeto de estudio, como caso de evaluación y adaptación de los resultados.

1.5 Método de Investigación

El método de investigación adoptado para este Trabajo Final de Maestría ha sido una adaptación del propuesto por Marcos y Marcos [18], [19] específica para la investigación en ingeniería del software. Dicho método se basa en el hipotético-deductivo presentado por Bunge [20], y se compone de una serie de pasos que se muestran en la Figura 3.

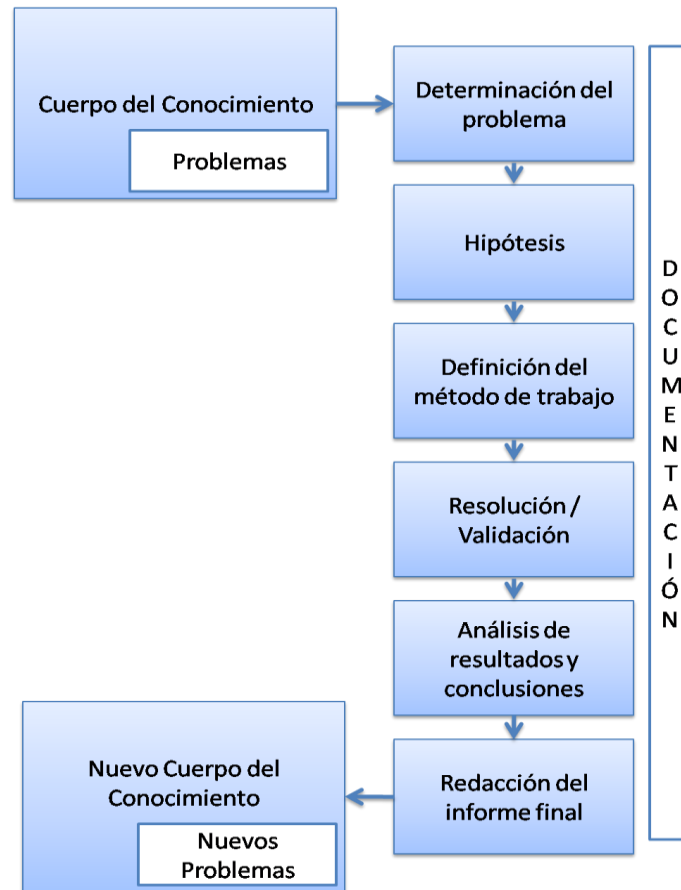


Figura 3 Pasos del método de investigación científica

La recopilación de información relacionada al estado del arte se realizó mediante revisiones sistemáticas de la literatura. La elaboración de la metodología y su validación en un conjunto de herramientas tecnológicas se llevó a cabo utilizando la aproximación de las metodologías ágiles. Las mismas, al ser iterativas e incrementales, permitieron someter las diversas etapas a procesos de mejora continua según los resultados parciales que se iban obteniendo en los ciclos de implementación y validación.

1.5.1 Etapa de Documentación

El método de revisiones 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 [28]. 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 Figura 4.



Figura 4 Fases del proceso de revisión sistemática de la literatura

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 extraer la información más relevante de cada estudio.

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

Se debe mencionar que existe 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 Figura 4, 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, la planificación. 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 el capítulo 2.2 Revisión Sistemática de la Literatura, se presentará la revisión sistemática realizada conforme al proceso que se acaba de describir.

1.6 Estructura del Trabajo

Este Trabajo Final de Maestría se encuentra organizado en cinco capítulos. Cada uno se describe de la siguiente manera:

- El capítulo 1 realiza una introducción al trabajo realizado, brindando una fundamentación de la propuesta y detallando los objetivos de la misma. Adicionalmente establece el área de conocimiento involucrada así como el marco y método de investigación.
- El capítulo 2 conforma el marco teórico que sustenta el trabajo realizado, explicando inicialmente el Desarrollo Continuo de Software, exponiendo mediante una revisión sistemática de la literatura el estado actual en torno a prácticas continuas. Finalmente, describe la forma en que el Sistema de Información Universitaria lleva adelante el desarrollo de software.
- El capítulo 3 expone la importancia de las prácticas continuas y las pruebas de software en el desarrollo, describe la situación actual del desarrollo de software en el SIU y realiza el planteo de la solución propuesta. Finalmente describe en detalle el modelo de procesos para pruebas continuas en el SIU.
- El capítulo 4 presenta la verificación del modelo de procesos para pruebas continuas propuesto. Inicia con la identificación del ecosistema software que permite dar soporte al modelo. Posteriormente, detalla la preparación requerida en el proyecto SIU-Araí: Usuarios y la implementación del modelo como caso de validación. Finalmente, presenta los resultados obtenidos y las discusiones sobre los mismos.

- Finalmente, el capítulo 5 presenta las conclusiones del trabajo, analizando el cumplimiento de los objetivos, con un contraste de los resultados obtenidos y presentando líneas de trabajo futuro.

Complementan este trabajo los siguientes anexos:

- El Anexo A: explica la tecnología Docker, el concepto de contenedor y su orquestación. Describe cómo usar la tecnología para empaquetar, distribuir y ejecutar aplicaciones.
- El Anexo B: describe el uso de Gitlab como plataforma para CI/CD, soportando el desarrollo continuo de software. Describe el flujo de trabajo y sus principales funciones.

2

MARCO TEÓRICO

2 **MARCO TEÓRICO**

En este capítulo se describe el estado del arte de los métodos de desarrollo continuo de software y las prácticas utilizadas. Se presenta una revisión sistemática de las diferentes prácticas de integración continua. Finalmente, se presenta un análisis de como el SIU lleva a cabo las pruebas de software.

2.1 **Desarrollo Continuo de Software**

La industria del software adopta paradigmas en el cual la funcionalidad del software está en continua evolución [6]. Esto es cubierto por el enfoque de la ingeniería del software conocido como Desarrollo Continuo de Software (del inglés Continuous Software Development, CSD) [7]. Su aplicación redundante en beneficios, pero también acarrea nuevos desafíos [8]. El CSD está enfocado en producir software en ciclos de tiempo reducidos, con su respectivo lanzamiento, manteniéndose así en un ciclo continuo. Involucra actividades de pruebas, actualizaciones, liberar versiones y medir aplicaciones de manera continua, ya que están en desarrollo. Tim O'Reilly [21] describe en su libro como los principios y prácticas de la comunidad Open Source inspiraron el desarrollo de software de la actualidad.

La Integración Continua (del inglés Continuous Integration, CI), las Pruebas Continuas (del inglés Continuous Testing, CT), el Despliegue Continuo (del inglés Continuous Deployment, CDE), la Entrega Continua (del inglés Continuous Delivery, CD), forman parte de las técnicas que surgieron en CSD [9], [10] para dar soporte a las actividades que significan producir software entregable, útil y verificado, en un tiempo razonable. Una de las premisas de CSD es lograr la entrega de valor al usuario final: debe ser en tiempos aceptables, pero también con un nivel de calidad y confiabilidad determinado, manejando los riesgos inherentes que pueden presentarse [8], [11], [12].

2.1.1 Integración Continua

La integración de 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. 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 [22]. La integración continua (del inglés Continuous Integration, CI) es una de las prácticas utilizadas para resolver estos inconvenientes.

El término “integración continua” quizás fue mencionado por primera vez en 1995, en el libro de Grady Booch [23]. En el mismo, expresa que “el proceso macro del desarrollo orientado a objetos, consiste en un proceso de integración continua”. Además, menciona que, en intervalos regulares de tiempo, el proceso de CI produce nuevas versiones del software ejecutables, que van creciendo en funcionalidad. Finalmente, Booch afirma que, gracias a estos hitos, se puede medir el progreso y la calidad, y de esta manera, anticipar, identificar y luego atacar los riesgos de forma continua.

Al principio, la práctica de ejecutar construcciones de código nocturnas automatizadas (nightly builds) ha sido considerada como la mejor solución durante muchos años. Sin embargo, otras prácticas similares han sido propuestas en diferentes libros y artículos, como [24] que recomienda la práctica de ejecutar construcciones diarias y pruebas de humo como parte de los proyectos de desarrollo de software.

Con la llegada de las metodologías ágiles, y con CI como una práctica recomendada, las personas comenzaron a notar que la construcción no debería ser diaria, sino “continua” [25]. Finalmente, Martin Fowler describe a CI de la siguiente manera [26]:

“CI es 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”

2.1.1.1 El proceso de integración continua

Un proceso de CI comienza cuando un desarrollador realiza un **commit**¹ de sus cambios hacia el repositorio donde se encuentra el código fuente. Sin embargo, este *commit* no solo se limita a código fuente del programa principal, sino que también abarca a cambios en esquemas de bases de datos, archivos de configuración, especificaciones, etc. La Figura 5 muestra un escenario completo de CI y sus componentes. El proceso de CI, abarca una serie de etapas [26]: 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 *commit*, y sus etapas son ejecutadas como parte del script de construcción. Es por ello, que el script de construcción tiene que ser capaz de ejecutarse con solo una línea de comando.

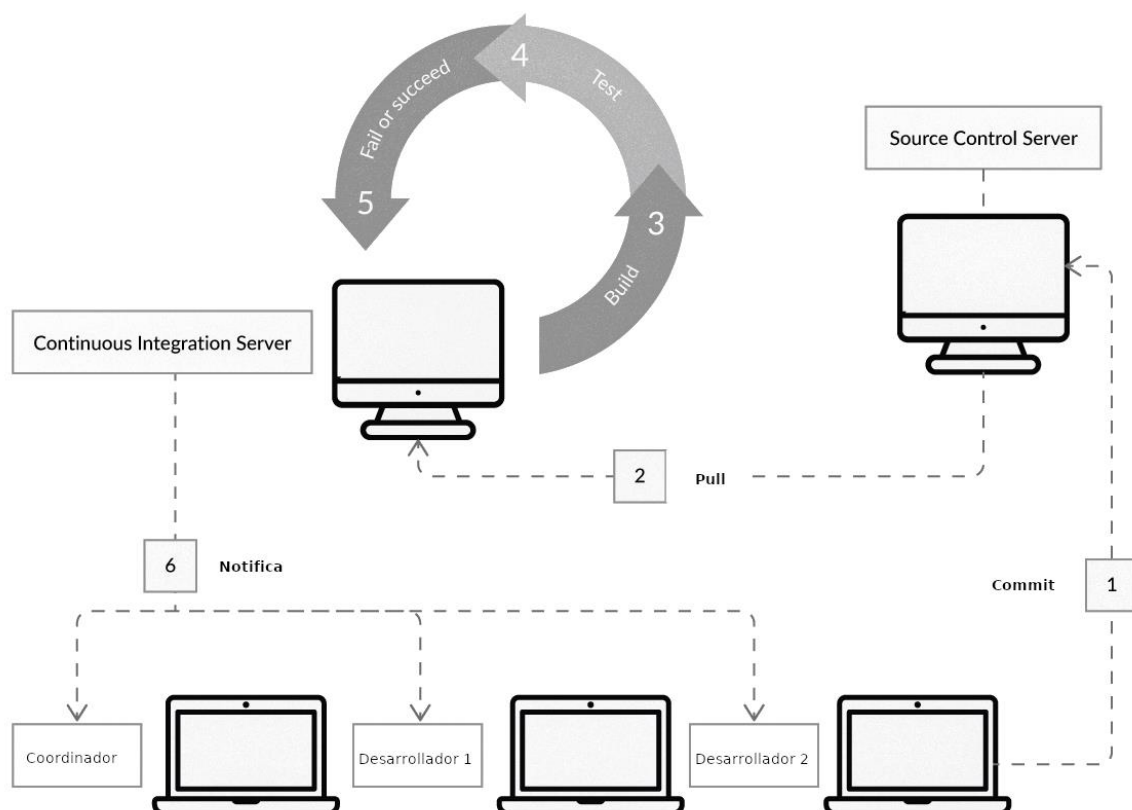


Figura 5 Escenario de integración continua y sus componentes

- 1 Commit: acción de confirmar un cambio e introducirlo a un sistema de control de versiones.

2.1.1.2 Componentes de la Integración Continua

En un proceso de CI, como se muestra en la Figura 5, se pueden identificar los siguientes pasos:

1. Un desarrollador introduce un cambio en el producto que desarrolla, realizando un **commit**, que almacena dicha modificación en el repositorio de control de versiones.
2. El servidor de CI detecta el cambio y se descarga el producto, realizando un **pull**.
3. El servidor de CI inicia la ejecución del proceso de construcción denominado **build**.
4. El servidor de CI ejecuta las pruebas que hayan sido definidas
5. Una vez finalizado la ejecución, el servidor de CI registra los resultados de la integración.
6. El Desarrollador es notificado por el servidor de CI con los resultados de la ejecución del proceso de integración.

En un proceso de CI se identifican los siguientes actores y componentes involucrados, presentes en la Figura 5:

- Desarrollador: la persona que modifica el código fuente y realiza el *commit*. Sin embargo, antes de enviar los cambios al repositorio de control de versiones, se recomienda que ejecute una construcción privada en su entorno local [27], 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 CI, 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) como por ejemplo Github² o Gitlab³.
- Servidor de CI: ejecuta la integración cuando nuevos cambios son introducidos en el repositorio de control de versiones. Normalmente, el

2 Github, <https://github.com>

3 Gitlab, <https://gitlab.com>

servidor de CI está configurado para detectar la introducción de estos cambios cada cierto periodo de tiempo o sobre determinadas acciones. El servidor de CI obtiene el código fuente y ejecuta el script de construcción. Asimismo, los servidores de CI pueden ser configurados para ejecutar el proceso de CI con cierta frecuencia (como por ejemplo cada hora) en lugar de realizarlo con cada cambio introducido. Existen una gran variedad de herramientas para operar como servidores de CI entre los que se encuentran Jenkins⁴, Bamboo⁵, Gitlab CI⁶ y TravisCI⁷.

- Script de construcción: puede ser uno o varios scripts, que se utilizan para compilar, probar, inspeccionar y desplegar el software. No necesariamente se usan herramientas, es posible realizar una construcción con órdenes mediante un script tipo Bash⁸ o Powershell⁹. Maven¹⁰, Rake¹¹, son ejemplos de herramientas que sirven para crear un script de construcción, pero no proveen CI por sí mismas.
- Sistema de Feedback: Uno de los principales propósitos de CI es producir retroalimentación o *feedback* al realizar las integraciones, ya que esto logra identificar la introducción de errores con el último *commit*. El propósito del *feedback* es crear un sistema de notificación que genere una acción lo más rápido y preciso posible [28]. Además, la información tiene que ser la correcta, a las personas indicadas, en el momento exacto y utilizando el medio correcto. CI es una herramienta que permite hacerlo, y en consecuencia, el *feedback* es continuo. El sistema de *feedback* permite enviar mensajes a los responsables de la construcción, con los resultados de la misma, tan pronto como finalice el proceso. De este modo, brinda a los mismos, la posibilidad de solucionar cualquier inconveniente que surja con rapidez. Otra característica es que este *feedback* debe estar disponible para su consulta a posterior, dando la posibilidad de tener un histórico de los resultados.

4 Jenkins, <https://jenkins.io>

5 Bamboo, <https://www.atlassian.com/software/bamboo>

6 Gitlab CI, <https://about.gitlab.com/product/continuous-integration/>

7 TravisCI, <https://travis-ci.com>

8 Bash, <https://www.gnu.org/software/bash/>

9 PowerShell, <https://docs.microsoft.com/en-us/powershell/>

10 Maven, <https://maven.apache.org>

11 Rake, <https://ruby.github.io/rake/>

2.1.1.3 Flujo de Integración Continua

Como se mencionó anteriormente, la Integración es parte del proceso de CI que abarca una serie de etapas [26]: compilación del código fuente, integración de la base de datos, ejecución de las pruebas, ejecución de inspecciones, despliegue del software y pruebas sobre el software desplegado.

La **compilación del código fuente** [29] es una de las etapas más comunes en un proceso de CI. Incluye la creación de código ejecutable a partir del código fuente. Esta etapa es la primera dentro de todo el flujo de CI. Luego de que el código fuente es compilado, se realiza la integración de la base de datos. La base de datos es una parte de toda la aplicación, y por ende, su integración, debe ser parte del proceso de CI. Ejemplos de scripts de integración de base de datos son: scripts de eliminación y creación de bases de datos o tablas, scripts para la inserción de datos de pruebas, scripts para aplicar procedimientos y disparadores (triggers), etc.

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. Del mismo modo, 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 complejidad, acoplamiento, cohesión, dependencia, cobertura, etc. [30].

Una vez que los cambios introducidos pasaron por todas las etapas mencionadas, se realiza un **despliegue del software** a un ambiente, 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 Figura 6. Mientras las primeras tres etapas deben ser tareas automáticas, el despliegue y la ejecución de pruebas siguientes pueden ser manual como automáticas. Sin embargo, para implementar despliegue continuo, el despliegue también debe ser automático.

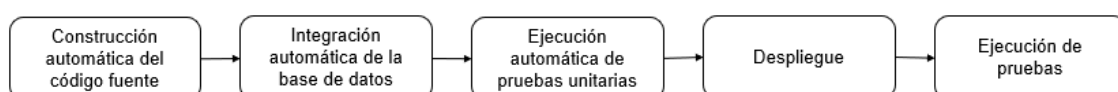


Figura 6 Flujo de actividades en la integración continua

2.1.2 Despliegue Continuo

El producto software debe llegar a los usuarios dentro de un periodo de tiempo establecido, ya sea por el cliente que lo solicitó, o por las necesidades de la compañía que lo desarrolla para competir en el mercado. El proceso de hacer que un software funcione en un determinado ambiente se denomina despliegue. Este ambiente no necesariamente es el entorno final donde los usuarios lo van a operar, puede ser un ambiente de desarrollo o de prueba. Sin el despliegue, el software no existe.

En la actualidad, el software se produce y se lanza al ambiente productivo con mayor frecuencia, y es por ello que el proceso de despliegue es tan importante como el de desarrollo. No solo es necesario integrar el software de manera frecuente, sino también desplegarlo a producción, es decir, se necesita tanto de un proceso de CI como de uno de Despliegue Continuo (del inglés Continuous Deployment, CDE).

La idea de desplegar el software de forma inmediata a producción no es reciente [31]. Sin embargo, utilizar esta práctica como una extensión de CI es una propuesta reciente [25], [32]. Estos definen a CDE como “una culminación de prácticas y pasos que permiten lanzar software funcional en cualquier momento y lugar, con el menor esfuerzo posible”.

Los principios de CDE se encuentran en los procesos de desarrollo de software ágil y lean, como se muestra en la Figura 7.

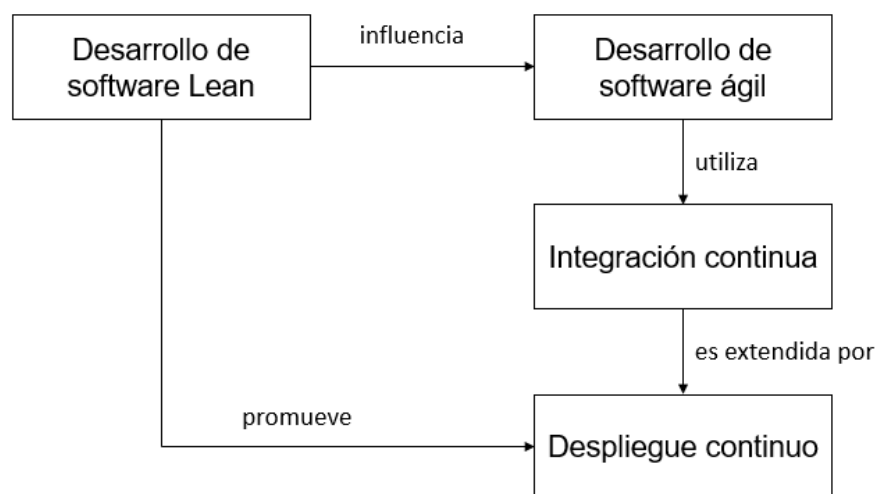


Figura 7 Influencia de los principios de desarrollo de software

La tendencia actual de las metodologías de desarrollo de software utilizadas en la industria es el desarrollo de software ágil [33]. 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” [34]. Si bien los desarrolladores de software ágil buscan comprender cómo entregar software de trabajo con frecuencia, ha habido poca investigación sobre cómo utilizar metodologías y procesos ágiles para lograrlo [12]. El proceso de CDE 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 lancen a producción más rápidamente.

Por otro lado, el desarrollo de software Lean (LSD) es otro de los promotores de CDE. LSD se define como “la aplicación de los principios del sistema de desarrollo de productos de Toyota al desarrollo de software” [35]. Proponen siete principios que forman la base del LSD: 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. Algunos especialistas han adoptado el uso de LSD [36], pero según Claps et al. [12], ninguno de ellos ha estudiado de forma empírica la efectividad de los principios propuestos por Poppendieck y Poppendieck [35]. Estos especialistas simplemente afirman que el LSD ha sido beneficioso para sus equipos de desarrollo de software. A pesar de esta falta de evidencia, CDE parece adherirse fuertemente a cuatro de los siete principios de LSD [12]: eliminar el desperdicio, ampliar el aprendizaje, entregar lo más rápido posible y empoderar al equipo.

De esta manera, los principios de LSD en conjunto con los métodos ágiles ayudan a los equipos de desarrollo a “entregar software lo más rápido posible” [12]. De acuerdo a Duvall et al. [25], 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 [48].

2.1.2.1 Despliegue tradicional y despliegue continuo

Lo que diferencia a CDE de los despliegues de software tradicionales, es la

frecuencia de los despliegues [12]: en CDE 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 1.

Tabla 1: Diferencias entre el despliegue tradicional y el continuo

Característica	Despliegue tradicional	Despliegue continuo
Frecuencia de los despliegues	Puede ir de 1 a 6 meses [37], incluso más. La frecuencia ha ido aumentando, de meses (metodologías de cascada o proceso unificado) a semanas (metodologías ágiles) [38].	Diaria. Empresas como IMVU despliegan el software a producción hasta 50 veces al día usando CDE. Otro ejemplo es Facebook, que lo hace una vez al día [38].
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 [15].	Como los cambios son pequeños y los despliegues frecuentes, el riesgo es menor [12].
Retroalimentación del cliente o usuario final	Se lo obtiene luego de largos periodos de tiempo, que dependen de la frecuencia de los despliegues [37].	Se lo obtiene muy rápidamente, ya que los clientes y usuarios perciben los cambios de manera constante [39], [40].
Funciones o características que no se terminan de desarrollar	Según el Standish Group [41], solo el 69% de las funciones o características que son solicitadas al inicio del proyecto, son desarrolladas.	CD ayuda a disminuir el desarrollo de características innecesarias, debido a la obtención de <i>feedback</i> frecuente [35].

Las empresas evolucionan sus prácticas de desarrollo de software con el correr del tiempo. Normalmente, existe un patrón que la mayoría de ellas siguen en su evolución: La “escalera al cielo” [42]. En el mismo, las organizaciones que

desarrollan software pasan, en primer lugar, de un enfoque tradicional a uno ágil. Luego, comienzan a madurar en las prácticas ágiles, incorporando procesos como CI. Finalmente, una vez que CI es alcanzado correctamente, mientras pequeños componentes del software son presentados al cliente durante las reuniones de presentación o “demos”, el mismo comienza a interesarse por tener estos componentes funcionales en producción. Es allí donde comienza la adopción de CDE.

2.1.2.2 Etapas del despliegue continuo

El primer requisito para hacer un despliegue frecuente del software es implementar CI [25]: la construcción del software y ejecución de pruebas en todos los niveles, deben ser tareas automáticas y repetibles. Sin embargo, además de esto, independientemente de la plataforma, tecnología o dominio, el despliegue de software a cualquiera ambiente abarca las siguientes etapas [25]:

1. Etiquetado de los activos que serán desplegados: se realiza para facilitar la identificación y seguimiento de los activos que van a ser desplegados en producción, y agruparlos por una característica similar, como por ejemplo un componente en una página web. Activos no solo son archivos del código fuente, sino también esquemas de bases de datos, scripts de pruebas automatizadas, archivos de configuración, elementos multimedia, etc.
2. Generación de un ambiente limpio: un error común es desplegar el software en un ambiente que tiene instalado una versión diferente del sistema operativo, base de datos, o servidor de aplicaciones. Es por ello que es importante generar un ambiente de despliegue limpio, para tener todos los parámetros de configuración deseados. El script de generación de ambiente será el encargado de producir las siguientes capas:
 - a. Sistema operativo requerido.
 - b. Configuraciones del sistema operativo, como conectividad de red, usuarios, firewall, etc.
 - c. Servicios, como servidor de aplicación, servidor de base de datos, servidor de mensajería, etc.
 - d. Configuración del servidor.

- e. Herramientas de terceros (third-party).
 - f. Software personalizado adicional.
3. Etiquetado de los artefactos binarios: en primer lugar, el código en el repositorio requiere una etiqueta. Sin embargo, para diferenciar la versión del software que está funcionando en un ambiente determinado, es necesario etiquetar también el artefacto resultante de la construcción del código en el repositorio, es decir, los binarios. El etiquetado de binarios permite identificar problemas, y asociar soluciones a una determinada versión.
 4. Instalación del artefacto binario en el servidor correspondiente: una vez que se tiene el artefacto binario etiquetado, se lo mueve al ambiente generado en la etapa 2 y luego se lo instala para que comience a funcionar.
 5. Ejecución de todos los tipos de pruebas necesarias, en todos los niveles: mientras ciertas etapas en el desarrollo del software requieren de la ejecución de solo algunas pruebas determinadas (por ejemplo, las pruebas unitarias al inicio), cuando el software es puesto en marcha en un ambiente, todas las pruebas deben ser ejecutadas. Es fundamental ejecutar previamente pruebas que detecten problemas en el ambiente, antes de ejecutar las pruebas unitarias, de integración y funcionales. Si bien algunas pruebas se ejecutan de forma automática, se requiere una etapa de pruebas manuales ejecutadas por un equipo especializado de pruebas.
 6. Creación de reportes de *feedback*: los reportes de *feedback* deben ser generados para ver los resultados del despliegue, y por ende debe ser fácil de entender por cualquier integrante del proyecto, incluyendo los roles no técnicos.
 7. Si fuera necesario, ejecución del roll-back utilizando las etiquetadas en el repositorio de control de versiones: se debe tener la capacidad de poner en ejecución la última versión estable del software.

Para CDE, todas estas etapas deben ser ejecutadas con tan solo un comando, es decir, deben estar automatizadas. Existen herramientas para realizar despliegues automáticos, que se utilizan en combinación con herramientas de CI. También existen herramientas que permiten implementar tanto CI como CDE.

Estas herramientas se mencionan en la sección de entrega continua a continuación.

2.1.3 Entrega Continua

Cuando en el 2001 se escribe el manifiesto ágil, se sabía que era importante la entrega continua de software, pero algo muy difícil de lograr [43]. De hecho, el primer principio del manifiesto ágil expresa: “nuestra principal prioridad es satisfacer al cliente a través de la entrega temprana y continua de software con valor” [16].

La Entrega Continua (del inglés Continuous Delivery, CD), es un enfoque de la ingeniería de software en el cual los equipos construyen software funcional en ciclos muy cortos de tiempo, de tal manera que el mismo pueda ser lanzado a producción en cualquier momento [44], [8].

Una de las características más importantes de CD, es la confiabilidad que se tiene al lanzar una nueva funcionalidad o cambio en el software [43]. Por ese motivo, cada cambio introducido debe someterse a un proceso riguroso de pruebas. Así, cualquier introducción, modificación o borrado de una funcionalidad, es lanzada a producción cuando se lo desee con un riesgo muy bajo de que contenga errores.

2.1.3.1 Entrega Continua y Despliegue Continuo

Con CD, el despliegue a producción es automático, y se debe pasar previamente por etapas de construcción y diferentes tipos de pruebas. Es por ello que CD y CDE son términos que usualmente son considerados iguales [43]. Sin embargo, CDE consiste en enviar a producción cualquier cambio que el desarrollador integre al repositorio principal. En cambio, CD si bien permite hacerlo, la decisión final de ir a producción es una decisión de negocio y no una técnica. La Tabla 2 muestra las principales diferencias y similitudes entre estos dos enfoques.

Tabla 2: Diferencias y similitudes entre CDE y CD

Despliegue continuo (CDE)	Entrega continua (CD)
Se enfoca en el despliegue a producción en el tiempo más rápido posible.	Se enfoca en la confiabilidad de introducir cambios a producción.
La decisión de lanzar una versión del software a producción no depende del negocio.	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.	Una persona debe presionar un botón (ejecución de un comando) para hacer el despliegue a producción.
Es un complemento de CI.	Es un complemento de CI.

CD y CDE son complementarios a CI, pero no se pueden hacer al mismo tiempo. Sin embargo, para ambos, el despliegue debe realizarse de manera automatizada. La diferencia es que, en CDE, el despliegue es gatillado inmediatamente luego de la finalización de una etapa anterior (por ejemplo, pruebas). En cambio, en CD, el despliegue es gatillado cuando se presiona el botón de despliegue. Estas diferencias se ven en la Figura 8.

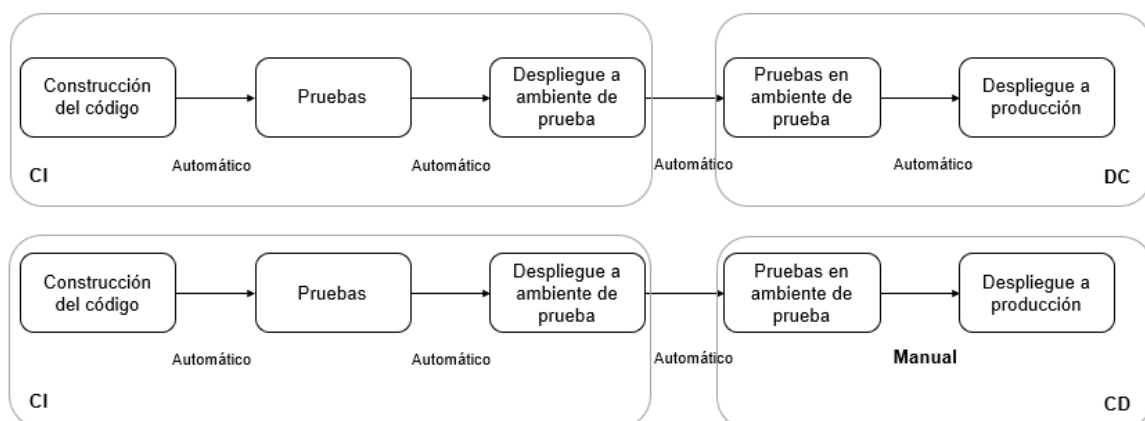


Figura 8 Diferencias en el pase a producción entre CDE y CD

Si bien en la Figura 8 la única diferencia apreciable es el despliegue a producción, las pruebas también son diferentes, ya que en CDE, se busca velocidad en el despliegue a producción, ejecutando solo las pruebas necesarias e involucrando al usuario final también como una etapa de pruebas más [45]. Por otro lado, en CD, se ejecutan la mayor cantidad de pruebas posibles asegurando que el producto a lanzar a producción esté libre de errores [43]. Sin embargo, en este último, la velocidad también es un factor clave y se busca hacerlo en el menor tiempo posible, ya que añadir más pruebas en el proceso lo hace más lento y, por ende, otros desarrolladores deben esperar más tiempo para introducir nuevos cambios [25].

2.1.3.2 Componentes de entrega continua

Uno de los principios de la entrega de software continua, es la creación de un proceso de despliegue del software a producción repetible y confiable [15]. Este principio se basa en que el lanzamiento de una versión del software a producción, debe ser una tarea sencilla, porque el mismo ya ha atravesado por diferentes niveles de pruebas. De esto modo, realizarlo “sería tan simple como presionar un botón” [15]. Para lograrlo, es necesario automatizar todas las tareas involucradas posibles y mantener todos los artefactos en el sistema de control de versiones.

Como se ha visto en CI y CD, la mayoría de las tareas son automatizables: compilación y construcción del código, diferentes niveles y tipos de pruebas, integración de la base de datos, despliegue a diferentes ambientes, generación de los ambientes de prueba, *feedback*, etc. [25]. Sin embargo, existen tareas que no pueden ser automatizadas. Entre ellas se encuentran:

- Ejecución de pruebas exploratorias realizado por un equipo de expertos y especialistas en pruebas.
- Demostración del software funcional a los responsables y dueños del producto.
- Aprobación de propósitos de conformidad.

En general y en cualquier tipo de proyecto, la lista de tareas que no se pueden automatizar es mucho menor que la lista de las tareas que sí se pueden

[15]. La automatización es un requisito indispensable para el proceso de CD, ya que es la única manera de garantizar que las personas podrán tener una versión del producto con solo presionar un botón.

Por otro lado, la gestión de la configuración es otro requisito fundamental para CD. Todo artefacto que se utilice para las tareas principales de construcción, despliegue y prueba debe ser parte del sistema de control de versiones [15]: código fuente, documentos de requerimientos, casos de pruebas, casos de pruebas automatizados, *scripts* de configuración de red, *scripts* de despliegue, *scripts* de procesos en la base de datos, archivos de configuración de la aplicación, librerías, dependencias, documentación técnica, etc. Los artefactos producto del despliegue también deben ser versionados [15], [25], ya que si se produce un error en el despliegue o en cualquier momento del flujo, la detección del artefacto y la versión que lo causó debe ser simple de identificar [43]. Es por eso, que los nombres de los *commits* deben ser claros y representativos de los cambios que se están introduciendo en el repositorio [15].

Otro factor relacionado con los despliegues del software es la infraestructura. En la mayoría de las organizaciones existen equipos de desarrollo y equipos de infraestructura u operaciones. Si las tareas relacionadas al manejo de infraestructura y ambientes son realizadas por ambos equipos de manera colaborativa desde el inicio del proyecto, serán llevadas a cabo más fácilmente [15]. Este foco en la colaboración es uno de los principios centrales del movimiento DevOps, el cual tiene como objetivo agilizar los procesos de administración de sistemas y operaciones.

DevOps es un cambio organizativo en el que, en lugar de existir grupos aislados divididos que realizan funciones por separado, los diferentes roles (sobre todo de desarrollo y operaciones) trabajan conjunto para lograr entregas continuas de funciones operativas [46]. Este enfoque ayuda a entregar valor de forma más rápida y continua, reduciendo los problemas debido a la falta de comunicación entre los miembros de los diferentes equipos y acelerando la resolución de problemas. Inclusive, algunas empresas adoptaron el término DevOps como un nuevo rol en la organización, refiriéndose a especialistas que realicen estas tareas, pero siguen teniendo sus barreras con los desarrolladores [43]. Es por ello, que lo importante es mejorar la comunicación y colaboración entre los diferentes roles.

Por último, el componente fundamental y requisito de CD es CI: desarrolladores integrando código con frecuencia, con ejecuciones de pruebas sólidas y teniendo como prioridad máxima solucionar errores producidos en la integración.

2.1.3.3 Conducto de despliegue

CI es un enfoque que permite que proyectos grandes y complejos puedan desarrollar el código de la aplicación con confiabilidad, ya que el mismo atraviesa por distintas etapas: construcción, compilación, pruebas, etc. [25]. Sin embargo, CI no es suficiente ya que la salida del mismo, es la entrada a una nueva etapa de pruebas y luego al resto del proceso de despliegue a producción [15]. De este modo, se pierde tiempo en esas etapas finales, por ende, el software tarda en llegar a un ambiente similar a producción, y en consecuencia a producción. Finalmente, el proceso de *feedback* entre estas etapas, al ser aisladas, no es inmediato. El conducto de despliegue o tubería de despliegue (del inglés *Deployment Pipeline*, DP) es la solución propuesta por Humble y Farley [15], como un enfoque holístico, de extremo a extremo para CD.

El DP es un patrón de CD, que consiste en un proceso automatizado que abarca todas las etapas desde la integración de código en el repositorio de control de versiones, hasta que llega a manos de los usuarios finales [15]. Una manera gráfica de visualizar el DP es presentada en la Figura 9.

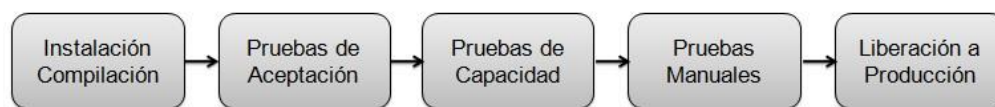


Figura 9 Conducto de despliegue, DP, propuesto por Humble y Farley

La entrada a DP se hace mediante la integración de código, donde el servidor de CI 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 o fallo en algunas de estas etapas, se genera *feedback* y se detiene todo el flujo.

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 DP depende de la necesidad de cada organización. La etapa que más varía en la implementación del DP es la de pruebas luego de los despliegues, variando en la cantidad de pruebas a las que se someten los cambios integrados. Los autores de CD, recomiendan las siguientes pruebas como necesarias en la implementación del DP [15]:

- 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 DP [47]. 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 CI, CDE y CD [9]. En la Figura 10 se muestran herramientas utilizadas en la implementación de DP, producto de una revisión sistemática de la literatura [9] general.

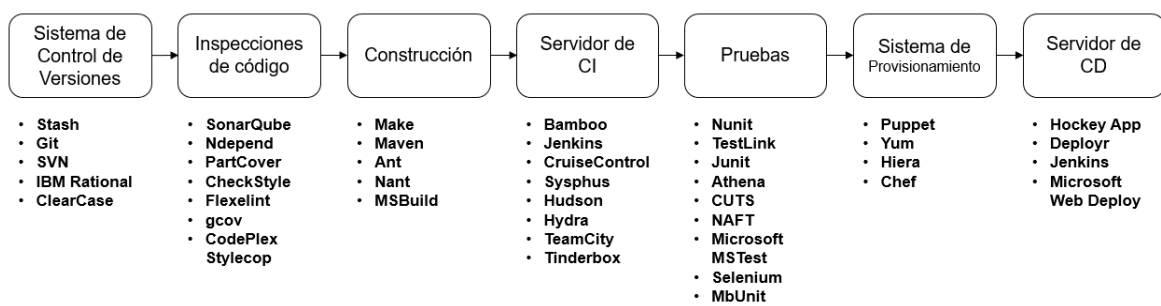


Figura 10 Herramientas típicas utilizadas para la implementación de un DP

2.1.3.4 Lanzamientos rápidos

Publicar una versión o realizar un lanzamiento de forma rápida es una práctica industrial que afecta varios dominios de la ingeniería de software, siendo parte de varias metodologías de desarrollo de software como *Lean*, *Agile* y *Open Source*. Según Mäntylä *et al.* [14] y Vesikivi [48], el despliegue automatizado es un facilitador para tener lanzamientos rápidos de versiones. Cuando hay menos cambios entre versiones, las actualizaciones pueden ser menos complejas y propensas a errores. Por el otro lado, requiere tener clientes que estén dispuestos a actualizar su software. Los lanzamientos rápidos pueden entrar en conflicto con el objetivo de tener alta fiabilidad [14].

2.1.4 Pruebas Continuas

Las pruebas de software son una actividad realizada para evaluar la calidad de un producto, busca mejorarlo mediante la identificación de defectos y problemas [17]. Para SWEBOK tiene cuatro características principales [17]:

- Dinámico: las pruebas se realizan solo sobre el programa en ejecución.
- Finito: las pruebas exhaustivas llevan mucho tiempo, es por ello que los ciclos de pruebas deben ser reducidos buscando cumplir un objetivo particular.
- Adecuado: existen muchas técnicas de pruebas, se deben elegir la más adecuadas para cada necesidad.
- Esperado: si no se cumple con una condición dada, los resultados indicarán 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 no es un proceso que comienza luego de la etapa de desarrollo, sino más bien una etapa complementaria [17]. 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 [17], [25], [49], [50].

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], [52], [53]. Una de estas normas, la ISO/IEC 25010 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 [54], [55], pero no todos pueden abordarse del mismo modo en los ambientes ágiles [49], [50]. 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 [15], mientras que Shahin et al. realizaron un estudio que demuestra que la automatización de pruebas es una pieza clave en CD [9]. Sin embargo, las pruebas exploratorias manuales son importantes en CD, no pueden ser automatizadas ya que requieren de la experiencia de un equipo especialista en pruebas manuales calificado [15], [8], [9], [49], [50].

A continuación se describen diversos tipos de pruebas de software propuestos para su ejecución en ambientes de desarrollo continuo.

2.1.4.1 Análisis estático de código fuente

El análisis estático del código fuente [56] es una de las primeras etapas de verificación de calidad en los entornos de desarrollo continuo [15], [25]. 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 particular (branch), o al tronco principal (trunk).

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” [57]. 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 [25]:

- Las inspecciones de código manuales son propensas a errores y no son repetibles con facilidad.
- Las revisiones de código entre pares pueden ser subjetivas.
- Son más rápidas que las manuales.
- Las herramientas de inspecciones de código permiten configurarse 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 [58]:

- Sentencias de código fuente no comentadas (NCSS).
- Complejidad del código (cognitiva, ciclomática, de Halstead) - CC, MVG.
- Código duplicado.
- Código no utilizado.
- Validación de estilos.
- Dependencias Cíclicas (CDC).
- Acoplamiento aferente (Ca).
- Acoplamiento eferente (Ce).
- Número de módulos (NOM).
- Líneas de código (LOC).
- Líneas de comentarios (COM).
- Peso de las funciones.
- Fan-in, Fan-In concreteness, Fan-out visibility.
- Cobertura de pruebas unitarias.
- Número de pruebas unitarias sin ejecutarse.
- Número de pruebas unitarias que fallan.

También existen herramientas que integran la medición de todas las métricas de mantenibilidad, como por ejemplo SonarQube, Sonargraph o inFusion [59], siendo la primera de código abierto. Al igual que el resto de las herramientas mencionadas, 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 [59].

2.1.4.2 Pruebas Unitarias

Las pruebas unitarias son verificaciones de pequeñas partes del código, por lo general funciones, métodos, subrutinas, o partes de ellas. Según SWEBOOK, las pruebas unitarias verifican el funcionamiento de trozos de software de forma aislada, y que se pueden probar por separado [17]. 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 [54]. Sin embargo, otros autores utilizan el término pruebas de componentes para referirse a las pruebas de integración [25]. 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. Sin embargo, para que las pruebas unitarias sean parte de un entorno continuo, las mismas deben automatizarse [25]. Es decir, se debe utilizar un trozo de código de un programa para probar un trozo de código de otro programa. Para automatizar las pruebas unitarias, existe en el mercado una gran variedad de tecnologías y frameworks dependiendo de la plataforma y lenguaje de programación del código a probar. Algunos ejemplos son JUnit, NUnit, PHPUnit, CppUnit, entre muchos otros más.

Normalmente, las pruebas unitarias requieren mínima dependencias de otras clases, teniendo en cuenta que 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 mocks. Los mocks son

objetos simples que simulan servicios reales, como una llamada a una base de datos.

El aspecto clave de una prueba unitaria es no depender de dependencias externas, las cuales tienden a incrementar la cantidad de tiempo que tarda la prueba en etapas de configuración y ejecución [17], [25], [60]. 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 [15].

2.1.4.3 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 [54]. Las estrategias de pruebas de integración clásicas, como top-down (de arriba hacia abajo) o bottom-up (de abajo hacia arriba), se utilizan en metodologías tradicionales de software, donde es estructurado jerárquicamente [17]. En cambio, las estrategias de integración más modernas se basan en la arquitectura, lo que implica integrar los componentes o subsistemas de software basados en funcionalidades [17].

Las pruebas de integración se utilizan para verificar las características de adecuación funcional y compatibilidad, dentro de la norma 25010 [25]. La adecuación funcional se mide al realizar la integración entre diferentes módulos funcionales del mismo sistema. Por otro lado, la compatibilidad se mide al realizar la integración del software con otros sistemas, como sistemas de archivos, bases de datos, hardware, sistemas operativos, etc.

Al igual que las pruebas unitarias, para los entornos continuos las pruebas de integración también deben ser automatizadas [25]. De acuerdo a Jöngren [61], 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 frameworks de pruebas unitarias para automatizar las pruebas de integración. Otros autores como Humble y Farley [15], afirman que también pueden utilizarse herramientas de automatización de pruebas de aceptación para el mismo

propósito.

Las pruebas de integración automatizadas requieren de más librerías o dependencias, para permitir la interacción con sistemas externos [25]. Existen ocasiones donde es posible conectarse a los sistemas mediante APIs, pero estas no siempre son accesibles.

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, subrutinas, etc.), las mismas deben ser ejecutadas luego de que las pruebas unitarias hayan finalizado [15], [25], [61].

2.1.4.4 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, y por ende, es una de las últimas etapas de verificación del lado del equipo de desarrollo [25]. 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 Selenium¹² [15], [25]. 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 [62].

12 Selenium: es un conjunto de herramientas con varias características que permite la automatización de acciones en un sitio web.

2.1.4.5 Pruebas No funcionales

Es un tipo de prueba de aceptación que se utiliza para verificar requerimientos no funcionales, por ende, mide 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 que es 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 [15]. Sin embargo, por un lado, los autores de CI 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 CD solo se centran en las pruebas de capacidad [15]. Según Chen [63], “si bien las pruebas unitarias y las pruebas de aceptación se han estudiado ampliamente en CD, las pruebas de requerimientos no funcionales han recibido una atención considerablemente menor”. La capacidad de un sistema abarca dos características principales [64]:

- Rendimiento (performance): mide el tiempo transcurrido en el que un sistema procesa una transacción, en condiciones determinadas de trabajo.
- Transferencia de datos (throughput): 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 [15]:

- Pruebas de escalabilidad: verifica cómo varía el tiempo de respuesta de un número N de peticiones al sistema, al agregar más servidores o servicios.
- Pruebas de “longevidad”: también llamada pruebas de picos (spike testing), 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 (memory leaks)¹³ 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.

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 DP [15]. 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 humo¹⁴ y finalmente la ejecución de las pruebas de capacidad.

2.1.4.6 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 CD, este grupo de tareas pertenecen a la etapa de pruebas de aceptación de usuario, que abarcan pruebas exploratorias, pruebas de usabilidad y demostraciones (showcases) [15], [49], [50]. Esta etapa debe realizarse continuamente e incluirse en el DP [15].

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

14 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, del conjunto universal de pruebas funcionales.

Las pruebas exploratorias son 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 [65], [66]. 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 [15], [66].

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 [49]. Es una manera de asegurar que “los usuarios finales pueden realizar sus tareas” [49], [50].

Por otro lado, las pruebas de usabilidad deben incorporarse en los entornos continuos, para descubrir qué tan fácil es para los usuarios finales cumplir con sus objetivos con el software que se está desarrollando [15]. Según la 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” [67]. Existen herramientas que permiten automatizar algunos aspectos de las pruebas de usabilidad [68], pero ninguna de ellas puede automatizar el proceso total. Las características de calidad que miden las pruebas de usabilidad son [68]–[70]: 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 o showcases. 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 [71]. 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, ya que es la primera vez que se tiene el *feedback* del cliente, usuario final o dueños de producto [15].

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 detectados por las pruebas automatizadas, y validando que aporte valor para los usuarios finales [15].

2.1.5 Las Pruebas en el Desarrollo Continuo de Software

“Errar es humano, perdonar es divino”, es una famosa frase del poeta Alexander Pope [72], que Jyothi, Krishna y Rao [73], 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 [73]: “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, la Junta Internacional de calificaciones para pruebas de software (ISTQB), afirma que todo 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 [54]. “Un software que no funciona correctamente puede dar lugar a muchos problemas, incluyendo la pérdida de dinero, tiempo o renombre, daños personales o incluso la muerte” [54]. En este sentido, Kuhn, Wallace y Gallo [74], 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 [75], que fue producida por un defecto de software: una conversión numérica errónea. Finalmente, en [76] 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 porqué son tan necesarias las pruebas de software en cualquier proceso de desarrollo. Tradicionalmente, las pruebas eran consideradas una tarea que debía realizarse al final del ciclo de desarrollo de

software, justo antes de que el producto sea entregado a los usuarios, como se muestra en la Figura 11.



Figura 11 Ciclo de desarrollo de software tradicional

Con el correr de los años, se comprobó que la detección de errores al final del proceso, era muy costosa en comparación con la detección de errores en etapas tempranas [77]. En la Figura 12 se muestra como el costo de solucionar defectos aumenta con respecto a las fases de desarrollo.



Figura 12 Costo de solucionar defectos en el software

Además de los costos elevados que generan encontrar y solucionar defectos al final del proceso de desarrollo de software, Mike Cohn presenta una lista de razones por las cuales el modelo tradicional de pruebas no funciona [71].

- Es difícil mejorar la calidad de un producto que ya existe.
- Los errores son desapercibidos.

- El estado del proyecto es difícil de medir.
- Las oportunidades de obtener *feedback* se pierden.
- Las pruebas tienden a ser interrumpidas, reducidas o abandonadas, debido a ser la última actividad del proceso, y se tienen que cumplir los tiempos de entrega.

Por estas razones, la mayoría de los equipos migraron a enfoques de desarrollo iterativos y ágiles [71]. Por ejemplo, los equipos que trabajan con Scrum, hacen del proceso de pruebas un componente que acompaña al ciclo de vida del desarrollo de software, donde se trabaja en mejorar la calidad de los procesos y el producto de manera continua, desde el inicio del proyecto. De acuerdo a Jain y Joshi [78], las pruebas deberían ser vistas como un esfuerzo colaborativo en todas las etapas, en lugar de una tarea que se realiza solo al final del proceso de desarrollo. Este modelo de pruebas se lo conoce actualmente como pruebas ágiles [49]. En la Figura 13 se muestra un ejemplo de un proceso iterativo, que incluye a las pruebas en cada iteración.



Figura 13 Etapas en un proceso de desarrollo de software iterativo

No obstante, que las pruebas acompañen todo el proceso de desarrollo de software, no es el único factor importante. En los procesos de desarrollo continuo, las mismas deben ser automatizadas [26].

En CI, se recomienda automatizar las pruebas unitarias, de integración y de rendimiento, e incluirlas en los scripts de construcción [25]. Estos scripts luego

tienen que ser ejecutados de manera local por los desarrolladores y posteriormente en el servidor de CI, al introducirse los cambios en el repositorio de control de versiones. En CD, se busca automatizar más tipos y niveles de pruebas [15], e introducirlos en el DP como se muestra en la Figura 9.

Las pruebas automatizadas, forman parte de uno de los componentes principales de CD, ya que el objetivo del mismo es realizar despliegues de productos fiables [15]. 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 [79]. 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 CD, mencionando que todavía se encuentran inconvenientes relacionados a las pruebas en este tipo de enfoque. Empresas y casos de estudios, reportan que mantener la calidad de software resulta un desafío al implementar CD.

El reto principal consiste en encontrar el balance entre tiempo y exhaustividad de las pruebas, recomendado por Duvall, Matyas y Glover [25]. 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 *feedback*. Por otro lado, si las pruebas no son lo suficientemente exhaustivas, se pueden introducir defectos en el software, con consecuencias como los problemas mencionados anteriormente. Sin embargo, los resultados de las investigaciones de Mascheroni e Irrazabal [80], muestran que aún no hay una solución a este desafío, y que es evidente la carencia de modelos y herramientas para asegurar la calidad de software en los procesos de desarrollo continuo.

De acuerdo a Prusak [81], “la industria aún no ha cerrado el círculo cuando se trata de implementar un proceso completo de entrega continua”. Si bien la literatura contiene instrucciones y buenas prácticas de como adoptar CD, su implementación ha sido un desafío en la práctica [13]. Es por ello que, para algunos autores, las pruebas continuas (del inglés Continuous Testing, CT) son el elemento faltante en el proceso de Desarrollo Continuo de Software, que incluyen las prácticas mencionadas [81], [80], [82].

2.2 Revisión Sistemática de la Literatura

Desde su introducción en 2004 [83], la Revisión Sistemática de la Literatura (RSL) ha ganado adopción en la ingeniería de software. Es una manera rigurosa y planeada de encontrar evidencia empírica de la literatura. Kitchenham provee una guía para investigadores sobre cómo utilizar dicha metodología [84].

Como parte de este TFM, se ha llevado a cabo un *Estudio Terciario de la Literatura* [85] para identificar estudios secundarios o RSL que analizan aspectos de las prácticas continuas, así como de las pruebas en el desarrollo del Software. El objetivo es establecer el estado del arte en cuanto a métodos específicos que estén involucrados en la Integración Continua, así como determinar las dificultades actuales al momento de aplicar tales métodos.

Este trabajo identificó 14 estudios secundarios, publicados entre 2012 y 2017, que se enfocan en temas relacionados con las prácticas continuas. Se concluye que el método de Integración Continua es aplicable a proyectos heterogéneas, destacando que la Automatización de las Pruebas es el tópico con mayores dificultades al momento de su aplicación.

2.2.1 Introducción

Actualmente, las organizaciones prestan significativamente más atención y destinan más recursos para desarrollar y entregar software de mayor calidad a un ritmo cada vez más rápido [47]. Es así como las prácticas de Integración Continua (CI) se han hecho populares, entre otras cosas, como una de las prácticas ágiles en el desarrollo de software [86].

CI se aboca principalmente a integrar el trabajo en progreso, múltiples veces en un corto período de tiempo. Su importancia se ha incrementado notablemente debido a los beneficios asociados con su implementación [87]. Estos beneficios incluyen incremento en la productividad de los desarrolladores, mejora de la comunicación en el equipo, mayor frecuencia y predicción en la entrega de características del producto, entre otros.

Las Pruebas de Software es, por lejos, el método más utilizado para el control y aseguramiento de la calidad en una organización que desarrolla software, siendo éstas una parte importante del proceso de desarrollo [88]. Las

Pruebas de Software hacen posible obtener información acerca de las diferencias entre el comportamiento actual de un producto de software y el requerido al ser desarrollado [89]. Los costos y el tiempo involucrado en pruebas pueden ser controlados a través de la automatización de las pruebas, donde la ejecución de una prueba es ejecutada en forma automatizada por una herramienta software en vez de un humano.

El objetivo de esta RSL es contribuir a un mejor entendimiento de las soluciones de CI y los desafíos que enfrenta una organización al momento de su adopción. Para lograrlo, se realiza una RSL que permite evaluar los datos y la información de una serie de RSL existentes, denominándose el resultado final un *Estudio Terciario de la Literatura* [84].

Esta RSL está organizada de la siguiente manera: una visión general del método de investigación es presentado en la sección 2, describiendo objetivos y preguntas de investigación, así como el proceso de búsqueda y extracción de los datos; una exposición de los resultados con un análisis sobre los estudios involucrados en la sección 3; finaliza en la sección 4 con la presentación de las conclusiones a las que se llega en esta RSL.

2.2.2 Método de Investigación

En esta sección se describe la metodología adoptada para realizar la RSL. Se eligió ejecutar un Estudio Terciario debido a que existe un gran número de estudios tipo RSL que abarcan diversos temas relacionados con las prácticas continuas en el desarrollo de software. Para guiar este trabajo, se siguen las guías para realizar un estudio tipo RSL que fuera propuesto por Kitchenham et al. [90]. Este método permite de forma transparente y rigurosa identificar, evaluar y sintetizar toda la investigación disponible que resulte relevante para una pregunta o tópico de investigación.

A continuación, se plantea el objetivo y las preguntas de investigación, el proceso de búsqueda y selección de estudios, el proceso de extracción de datos y se finaliza con el método para evaluar la calidad de los mismos.

2.2.2.1 Objetivo y Preguntas de Investigación

El objetivo de esta investigación es conocer el estado de situación y realizar la búsqueda de evidencias sumariadas mediante RSL en torno a temas relacionados con CI. Este trabajo define cinco preguntas de investigación a las que dará respuesta:

Q1 ¿Cuáles son los tópicos de investigación en torno a la integración continua?

Q2 ¿Qué métodos relacionados con la integración continua se describen?

Q3 ¿Con qué dificultades se encuentran en torno a la integración y en particular a las pruebas continuas?

Q4 ¿Qué herramientas utilizan para dar soporte a las actividades de integración continua?

Q5 ¿Qué lenguajes de programación y tipos de soluciones están produciendo?

2.2.2.2 El Proceso de Búsqueda

Las búsquedas fueron realizadas en forma sistemática, utilizando como fuente de consulta los siguientes motores de búsqueda para identificar estudios a ser analizados: ScienceDirect, ACM, IEEEExplore y Scopus.

A continuación, se muestra la construcción de la consulta para el servicio Scopus (siendo adaptable a los otros motores de búsqueda mencionados previamente).

```
TITLE-ABS-KEY("evidence based software engineering" OR "review of studies" OR  
"structured review" OR "systematic review" OR "literature review" OR  
"literature analysis" OR "in-depth survey" OR "literature survey" OR "meta-  
analysis" OR "Past studies")  
AND TITLE-ABS-KEY("Continuous Development" OR "Continuous Integration" OR  
"Continuous Deployment" OR "Continuous Delivery" OR "Rapid Releases" OR "test  
automation")  
AND TITLE-ABS-KEY(software)  
AND ( LIMIT-TO ( SUBJAREA,"COMP " ) )
```

El proceso de ejecución de las búsquedas en cada uno de los motores seleccionados fue realizado de forma manual. Al cabo de su ejecución, se extrajo

información vital de cada estudio (buscador, título, autores, año, tipo de estudio, citas).

Los resultados obtenidos de cada buscador pueden ser vistos en el documento “resultado de las búsquedas” [91], en la página “búsqueda”. Los resultados fueron extraídos ejecutando las consultas en el período comprendido entre el 01/04/2018 y el 10/04/2018.

2.2.2.3 Selección de los Estudios

Para la selección de los estudios a analizar, los cuales se obtuvieron a partir de las búsquedas previamente realizadas, es necesario criterios de inclusión y de exclusión. El criterio de inclusión se definió de la siguiente forma:

- Estudios relacionados a alguna práctica de CI, CD, CDE en uso o aplicada al desarrollo de software

De forma complementaria, se estableció los siguientes criterios de exclusión:

- Estudios duplicados (mismo estudio publicado en diferentes lugares y/o figurando en los resultados de diversos motores de búsqueda)
- Estudios que no sean del tipo revisión sistemática de la literatura o estudios semi-sistemáticos de la literatura, ni estudios de mapeo
- Estudios que no respondan a ninguna de las preguntas de investigación

2.2.2.4 Proceso de Extracción de los Datos

Los estudios obtenidos por medio de las búsquedas realizadas en los motores mencionados anteriormente, fueron sometidos a un análisis de múltiples pasos, aplicándose los diferentes criterios de selección y luego de exclusión.

Paso 1: en primera instancia, mediante la lectura del título y el resumen (abstract), se aplicó a cada resultado de la búsqueda el proceso de “selección de estudios”. Con ello se filtraron los estudios que no tuvieran relación alguna con las técnicas CI-CD-CDE.

Paso 2: se obtuvieron copias completas de cada estudio. Se realizó una lectura completa para determinar si continuaban cumpliendo con los criterios de

selección.

Paso 3: por cada estudio, se extrajo al menos un concepto que se relaciona o ayuda a responder las preguntas de investigación de este trabajo.

En la Figura 14 se observa el proceso de cómo se han ejecutado las búsquedas, los resultados obtenidos y la cantidad de estudios que han sido procesados. Los datos de las búsquedas iniciales, los estudios seleccionados, así como los conceptos sin procesar pueden ser vistos en el documento “resultado de las búsquedas” [91], en sus diferentes páginas.

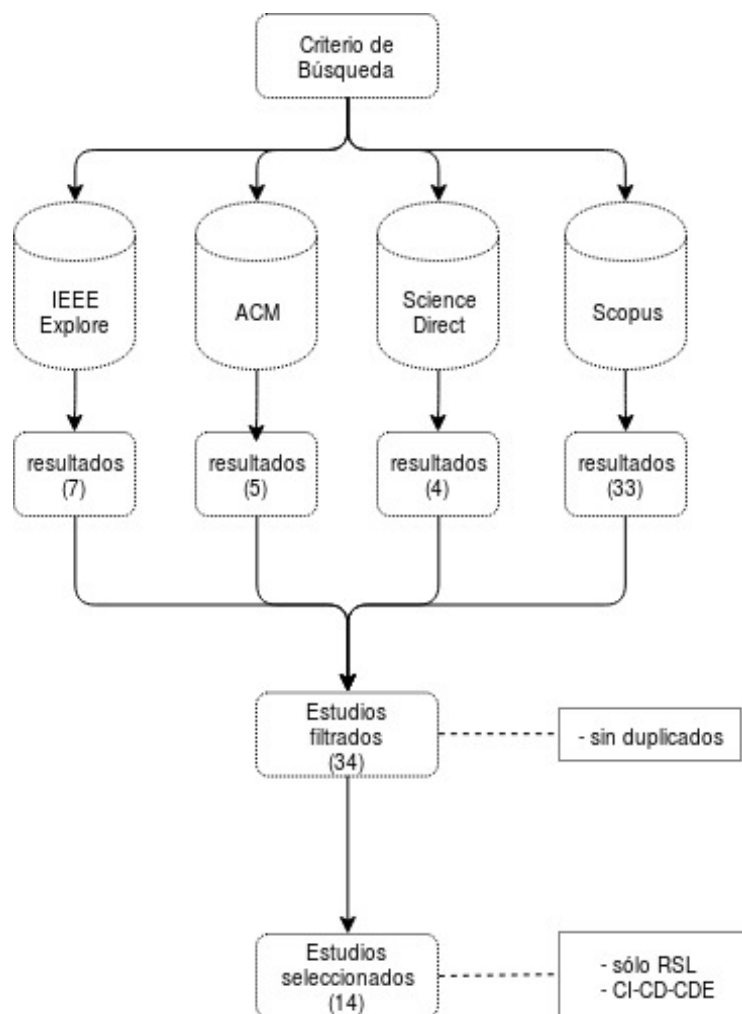


Figura 14 Proceso de búsqueda y selección de estudios

En resumen, para este trabajo se han seleccionado 14 estudios primarios tipo RSL como objetos de estudio, donde:

- 49 resultados fueron obtenidos de la búsqueda inicial en los cuatro motores mencionados
- 15 resultados fueron estudios duplicados, que aparecen en más de un motor
- 20 resultados fueron estudios no relacionados con CI-CD-CDE

2.2.2.5 Evaluación de Calidad

Cada RSL fue evaluada utilizando el criterio específico de la Base de Datos de Resúmenes de los Efectos (DARE) [92], definido por el Centro para Revisión y Disseminación de la Universidad de York. Este criterio está basado en cuatro preguntas:

1. ¿Los criterios de inclusión y exclusión están definidos y son apropiados?
2. ¿Es probable que la búsqueda en la literatura haya cubierto todos los estudios relevantes?
3. ¿Los revisores evaluaron la calidad / validez de los estudios incluidos?
4. ¿Fueron los datos / estudios básicos adecuadamente descritos?

Tabla 3: Escala de evaluación de los criterios DARE

N°	Sí	Parcialmente	No
1	Criterios explícitamente definidos	Criterios implícitos	Criterios no definidos, ni inferibles
2	Búsqueda en 4 o más bibliotecas digitales, incluyendo estrategias para buscar en revistas	Búsqueda en 4 o más bibliotecas digitales, sin otra estrategia adicional	Búsqueda en no más de 2 bibliotecas digitales, con un conjunto restringido de revistas
3	Criterios de calidad definidos y extraídos de los estudios primarios	Las preguntas de investigación involucran problemas de calidad que son respondidas por el estudio	Criterios de calidad NO definidos o extraídos de los estudios primarios, pero no utilizados.
4	Información de cada estudio primario es presentado y sumariada, la información es trazable hasta el estudio primario	Solo información sumariada, ej: agrupando en categorías sin posibilidad de enlazar de forma individual a cada estudio con la categoría.	Los resultados de los estudios individuales no son especificados, los estudios primarios no son citados

Las preguntas fueron puntuadas con un S, P o N, de acuerdo al criterio establecido en la Tabla 3.

2.2.3 Resultados

Esta sección reporta los resultados en relación a cada una de las preguntas de investigación. Se han encontrado 14 (catorce) RSL que abordan aspectos de prácticas continuas, como se muestra en la Tabla 4.

Tabla 4: Listado de estudios secundarios o RSL obtenidos para su análisis

Nº	Título de la publicación	Tipo	Año	Ref.
P1	Impediments for software test automation: A systematic literature review	SLR	2017	[93]
P2	The EMFIS model - Enable more frequent integration of software	SLR	2017	[94]
P3	Frequent releases in open source software: A systematic review	SLR	2017	[95]
P4	Choosing the right test automation tool: A grey literature review of practitioner sources	MLR MLR	2017	[96]
P5	Systematic literature review on the impacts of agile release engineering practices	SLR	2017	[97]
P6	Problems, causes and solutions when adopting continuous delivery - SLR	SLR	2017	[13]
P7	Continuous Integration, Delivery and Deployment: A Systematic Review on Approaches, Tools, Challenges and Practices	SLR	2017	[9]
P8	A model for determining the maturity of automation of software testing as a research and development area	SLR	2017	[98]
P9	When and what to automate in software testing? A multi-vocal literature review	MLR	2016	[99]
P10	On rapid releases and software testing: a case study and a semi-systematic literature review	SLR	2015	[14]
P11	Build Waiting Time in Continuous Integration-An Initial Interdisciplinary Literature	SLR	2015	[100]
P12	Modeling continuous integration practice differences in industry software development	SLR	2014	[87]
P13	Software paradigms, assessment types and non-functional requirements in model-based integration testing	SLR	2014	[101]
P14	Benefits and limitations of automated software testing: Systematic literature review and practitioner survey	SLR	2012	[102]

El análisis inicial de los resultados muestra que las revisiones sistemáticas en esta área son más bien recientes. Se pasó de una publicación en el año 2012 a ocho en el 2017. En la Figura 15 se observa la distribución de los resultados en el tiempo, donde se destaca este crecimiento exponencial en el último año, acompañando la expansión de la disciplina CI.

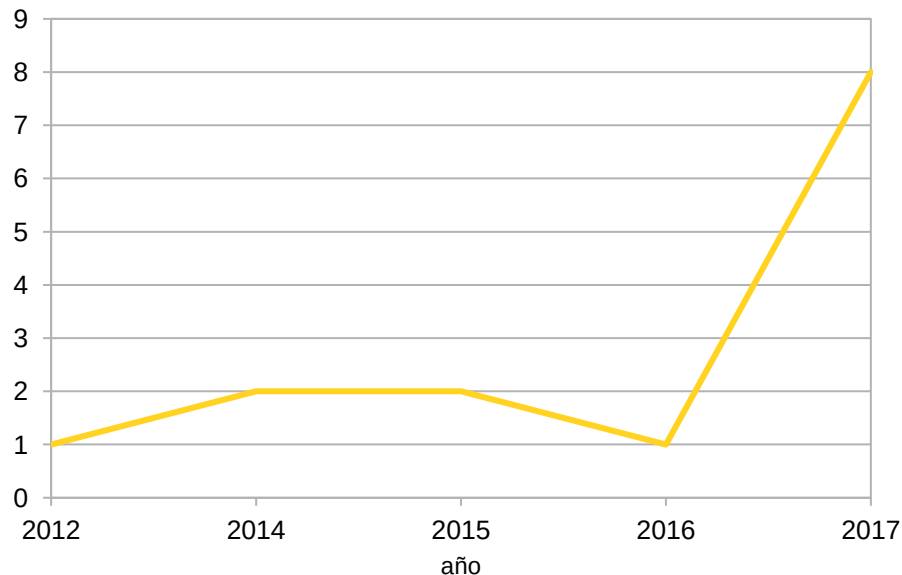


Figura 15 Distribución de las publicaciones a lo largo de los años

En las siguientes secciones se discuten las respuestas a las preguntas de investigación.

Q1 ¿Cuáles son los tópicos de investigación en torno a la integración continua?

Esta pregunta permite conocer las áreas que motivan la investigación de los trabajos relacionados. En la Figura 16 se listan los tópicos identificados en los estudios como los principales ejes de investigación que fueron encarados.

La *Ingeniería Continua de Software* pretende englobar todos los conceptos tales como la integración, despliegue y entrega continua. Tres cuartas partes de los estudios obtenidos analizan en general estos conceptos, proponiendo desde un modelo para analizar el estado de situación de la organización [94], hasta evaluando los retos y dificultades que se presentan adoptando este tipo de prácticas [13], [87], [95], [97].

La *Automatización de las Pruebas* es el tópico que mayor atención recibe

en los estudios analizados, representando la mitad de los mismos. Varios estudios exponen los beneficios y limitaciones [93], [99], [102] de esta actividad, así como la selección y uso de las herramientas para pruebas disponibles [14], [96].

El último tópico refiere a las *Prácticas, Aproximaciones, Herramientas y Desafíos* con las que dos estudios [9], [100] han analizado en detalle actividades ligadas estrechamente al desarrollador de software utilizando técnicas de CI-CD-CDE.

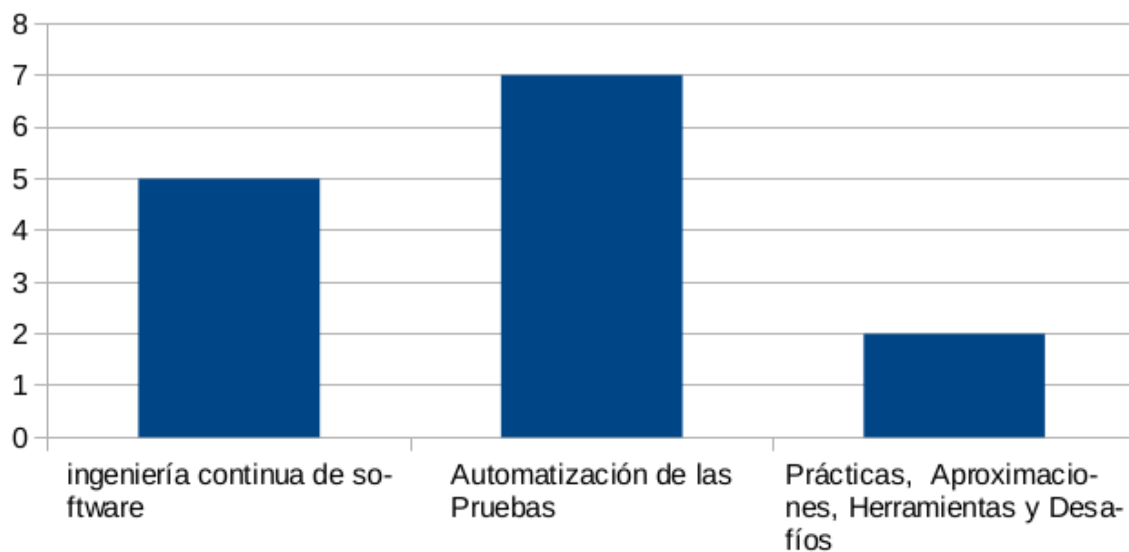


Figura 16 Tópicos de investigación analizados en torno a CI

Q2 ¿Qué métodos relacionados con la integración continua se describen?

Esta pregunta permite conocer si existen otros métodos, estrechamente relacionadas con la integración continua, expuestas en los estudios analizados. En la Figura 17 se listan todos los métodos identificados en los estudios analizados.

Las prácticas de *Pruebas Continuas* (junto con muchas técnicas afines y/o dependientes) se lleva la atención de más de la mitad de los estudios analizados. Esto destaca la importancia de las pruebas en la aplicación de la ingeniería continua del software. Los estudios hacen énfasis en los tipos de pruebas automáticas necesarias para aceptación de los cambios [93], así como para la integración [97], [101] y despliegue o puesta en producción [97]. En detalle, evalúan los impedimentos para su adopción tanto de la organización e individuos como de las herramientas involucradas.

Las prácticas de *Despliegue Continuo* (que requiere de la *Entrega Continua* e *Integración Continua*), representan la otra mitad de los estudios analizados, demostrando que dichas prácticas están estrechamente relacionadas. Se expone métodos sobre como realizan lanzamientos continuos proyectos de código abierto [95] así como el impacto del uso de las metodologías ágiles en dicha actividad [97]. Adicionalmente, los estudios evalúan los beneficios [87] así como también problemas [100] y desafíos [9], [13] al aplicar actividades relacionadas con la integración continua.

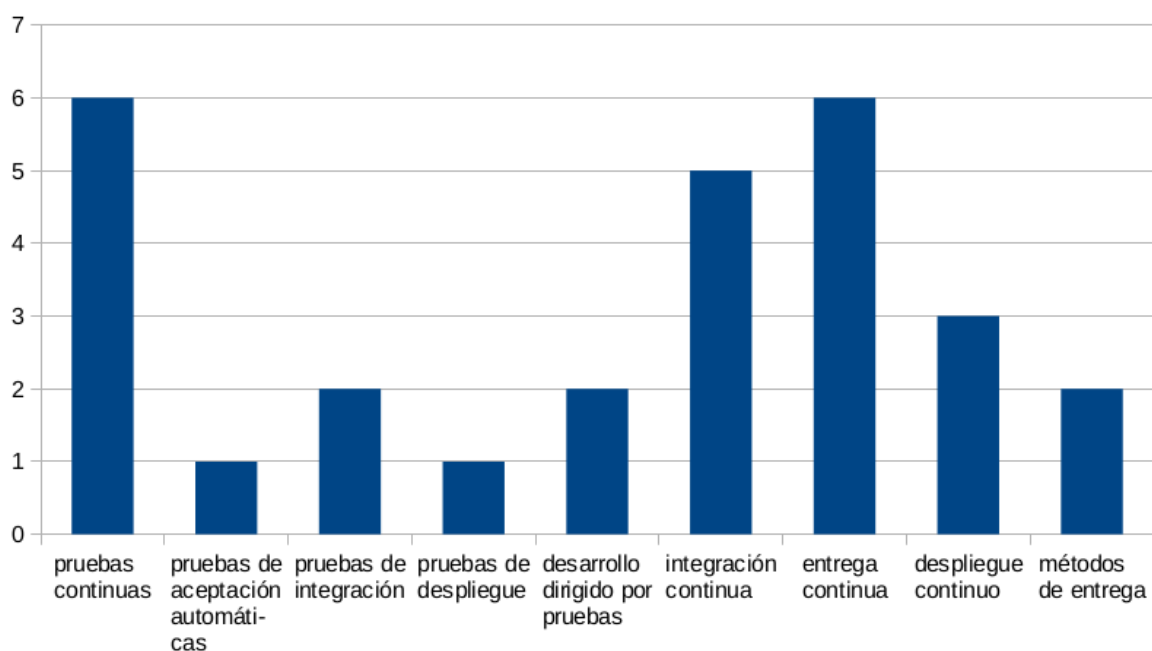


Figura 17 Métodos relacionados con las prácticas continuas

Q3 ¿Con qué dificultades se encuentran en torno a la integración y en particular a las pruebas continuas?

Esta pregunta tiene como objetivo conocer los inconvenientes que se presentan en la adopción de las prácticas de integración continua, con especial detalle en las pruebas, para los estudios que han sido analizados. En la Figura 18 se listan dificultades identificados en los estudios analizados.

Los Costos, tanto económicos como expresados en tiempo, representan un impedimento en la adopción de prácticas de integración [9] y en particular en la automatización de pruebas [93] [102]. La *Dificultad para Automatizar las Pruebas*, por otro lado, es uno de los factores de mayor preponderancia en los estudios

analizados. Algunos motivos son la arquitectura de los casos de pruebas, incremento de costos por la introducción de nuevos casos [99] y su integración a los flujos continuos [100].

La *Conducta de la Organización*, apoyados en el escepticismo a implementar prácticas continuas [9] así como en evaluar los beneficios de las pruebas [102], es un factor que debe ser controlado para evitar que impacte de forma negativa en actividades relacionadas a la integración.

La *Complejidad de la Aplicación*, fundamentada principalmente en una arquitectura de la aplicación que entorpece la automatización de las pruebas [93] así como la automatización y mantenimiento del proceso de construcción [13], representa una dificultad que provoca múltiples diferencias al momento de adoptar prácticas continuas [87].

La *Deuda Técnica* es otro factor de dificultad que, en los estudios analizados, se vincula a la arquitectura de la aplicación y la forma de implementar las pruebas [102].

Por último, la *Deficiencia en las Herramientas* [96] así como los *Flujos de Integración Continua*, son factores que sustentan la falta de homogeneidad en las prácticas continuas [87], lo que conlleva a mayores dificultades al momento de determinar un modelo a aplicar.

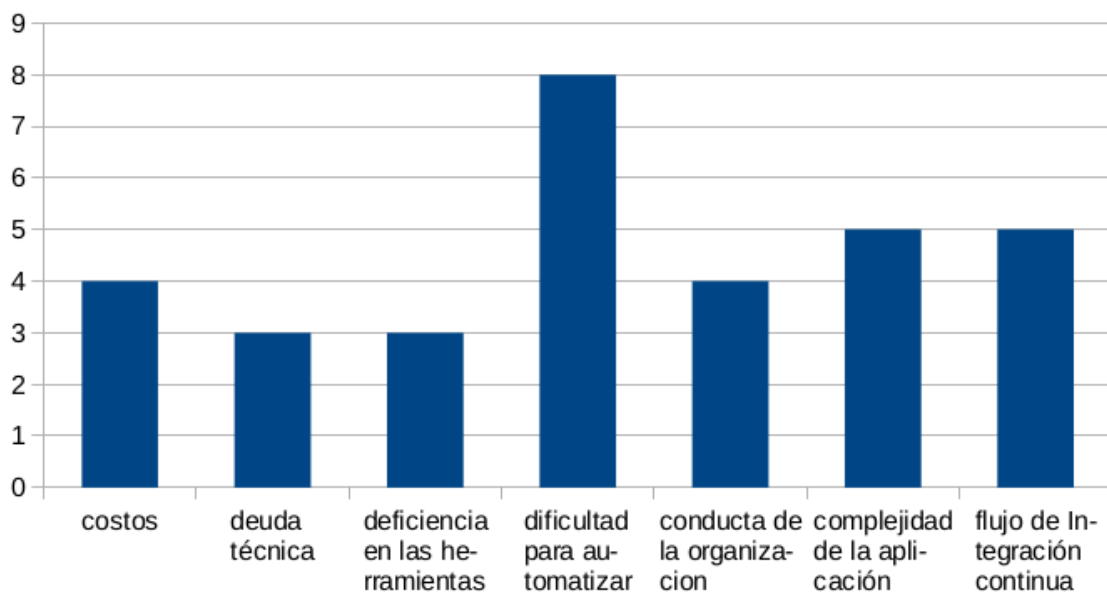


Figura 18 Dificultades relacionadas con la integración continua

Q4 ¿Qué herramientas utilizan para dar soporte a las actividades de integración continua?

Esta pregunta permite conocer qué tipos de herramientas están involucradas en las actividades de CI, de acuerdo a los estudios analizados, según la Figura 19. La mayoría de los estudios mencionan el uso de tipos de herramientas que permiten manejar el *Flujo de Integración Automatizada* (también conocido como **pipeline** [9]), realizando tareas como la construcción automática de la aplicación, la ejecución automática de las pruebas, asistir en la revisión del código fuente y facilitar la comunicación entre el equipo de desarrollo. Se dan ejemplos varios de productos actuales (Jenkins, Bamboo, SonarQube, TravisCi).

El otro tipo de herramientas que tratan los estudios analizados, está vinculado más específicamente a las pruebas. Partimos del tipo *Captura y re-ejecución de Acciones*, que permite generar casos de prueba de forma simple [93]; luego están las herramientas destinadas a *Generar Datos de Prueba*, para automatizar la generación de diversas variantes o caminos de prueba [101].

También se contempla que dichas herramientas puedan ser ejecutadas tanto desde un ambiente automatizado como desde un entorno local [95], [101], para obtener una devolución temprana. Se dan ejemplos varios de productos actuales (Selenium, MozTrap, scripts).

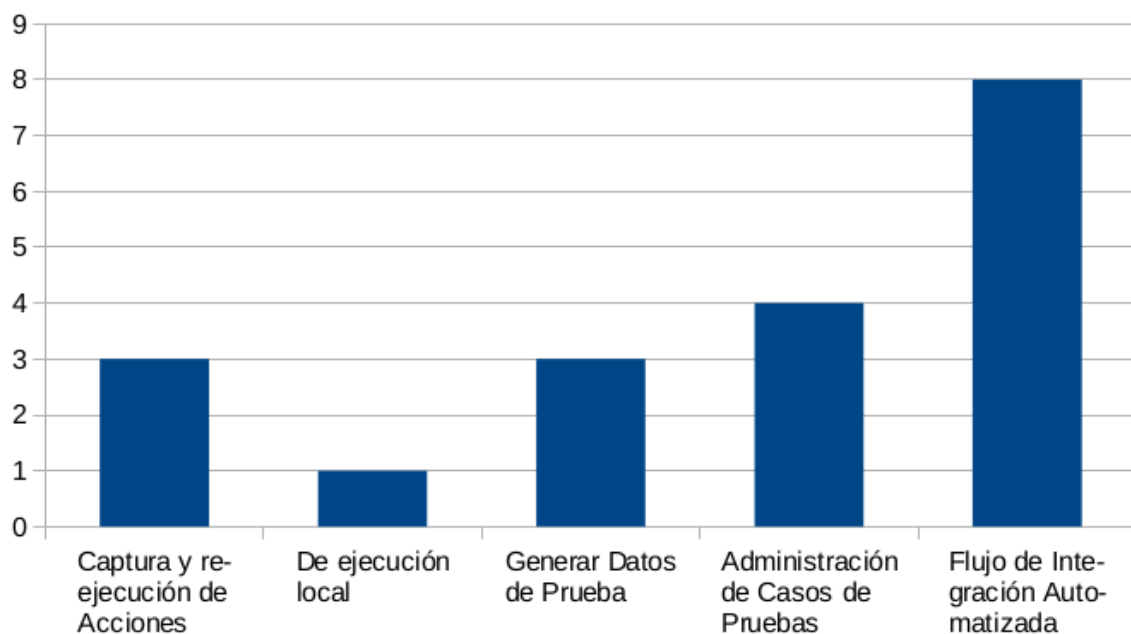


Figura 19 Herramientas utilizadas para dar soporte a las prácticas continuas

Finalmente, un tipo de herramienta necesaria es la destinada a la *Administración de los Casos de Pruebas* [9], permitiendo el seguimiento y explotación de la información de las ejecuciones de las pruebas [98], establecer y analizar diversas métricas vinculadas al retorno de la inversión [99]. Se dan ejemplos varios de productos actuales (TestLink, IBM RQM).

Q5 ¿Qué lenguajes de programación y tipos de soluciones están produciendo?

Esta pregunta permite conocer qué tipo y tamaño de aplicaciones se analizan en los estudios seleccionados, así como también el tipo de lenguaje de programación utilizado.

De los estudios analizados, en general no excluyen el tipo de aplicación al momento de aplicar prácticas continuas. Los tipos de soluciones que son analizadas son descritos como *Software General* [14], [95], [99], pero con particular detalle a *Software Crítico* [94] y *Componente Orientado a Objetos/Servicios* [101]. El tamaño del producto o aplicación también es mencionado en los estudios, siendo generalizado de *Pequeño a Grande* [9], [13], [14], [97]–[99], solo *Grandes* [94] u orientado a *Kits de Herramientas* [95], [96]. En la Figura 20 se representa esta distribución de valores.

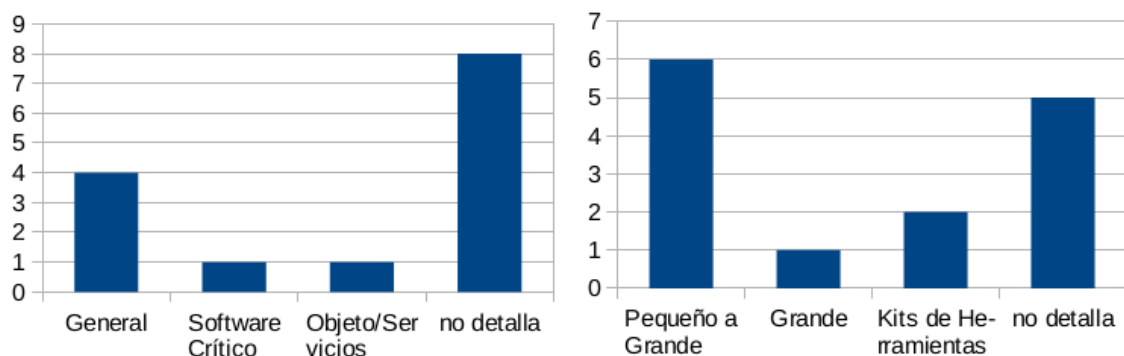


Figura 20 Tipos y tamaños de aplicaciones reportados en los estudios

Por último, el lenguaje de programación utilizado en los estudios analizados está mayormente descrito como *Genérico* (cuando es aplicable al tipo de producto que se construye) u orientado a *Aplicaciones de Escritorio* [14], y de

tipo *Scripting* (cuando es aplicable al manejo de herramientas [96]). No representa para los estudios analizados, un factor de discriminación las tecnologías utilizadas en el desarrollo del producto. En la Figura 21 se representa esta distribución de valores.

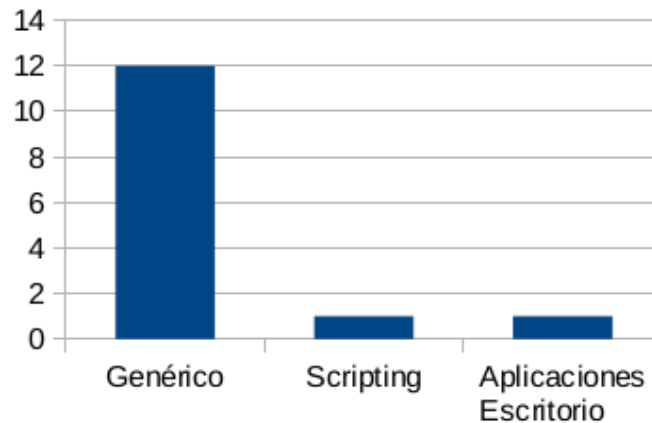


Figura 21 Tipos de lenguajes de programación

2.2.4 Conclusiones

Este trabajo provee una revisión sistemática del estado de situación en torno a las prácticas continuas. Si bien la disciplina no es nueva, apenas existen estudios secundarios entre el año 2012 y el año 2016. En cambio, con el año 2017 se produce un crecimiento exponencial de estudios secundarios o RSL. Esto implica que, dentro del desarrollo de Software, estas prácticas están recibiendo mayor atención tanto de la academia como de los practicantes en la industria.

Para todos los estudios analizados, uno de los tópicos más importantes representa la Automatización de las Pruebas. No es coincidencia que también se describe como una de las mayores dificultades que se reportan al momento de aplicar prácticas continuas. Según estos estudios, existen múltiples factores que lo sustentan: la conducta de la organización influyendo en la toma de decisiones, la arquitectura de la aplicación al momento de construir y ejecutar los casos de pruebas, así como los costes económicos de asumir dicha implementación inicial.

La Integración Continua, resulta una práctica habitual en el marco del Desarrollo Continuo de Software (existe una relación de dependencia con el Despliegue y la Entrega Continua). Según los estudios analizados, esta práctica

es aplicable a cualquier tipo de aplicación sin importar tamaño ni las tecnologías implicadas en el desarrollo. Es más, para dar soporte a la automatización de este tipo de flujos continuos, las herramientas utilizadas suelen ser genéricas y aplicables sin mayores restricciones de este tipo.

2.3 Pruebas de Software en el marco del SIU

El Sistema de Información Universitaria (SIU) se enfoca en el desarrollo de software de gestión capaz de mejorar la calidad y disponibilidad de los datos y fortalecer la gestión de las instituciones universitarias, en cada uno de sus ámbitos de acción [3]. Esta sección describe de forma concisa qué es el SIU, las actividades que realiza y de qué manera lleva a cabo el desarrollo de software, y en particular los tipos de prueba de software que realiza.

2.3.1 Breve historia

En la actualidad, el Sistema de Información Universitaria (SIU) [2] es una institución dependiente del Consejo Interuniversitario Nacional (CIN), órgano que nuclea a todas las universidades del sistema público nacional argentino. A su vez, el CIN está conformado como una persona de derecho público no estatal que se sostiene, primordialmente, con los aportes que realizan sus miembros.

Los inicios del SIU se remontan al año 1996 [103], cuando es creado el Programa SIU por la Secretaría de Políticas Universitarias del Ministerio de Educación de la Nación Argentina [1], con financiamiento del Banco Mundial. Por el año 2007 se conforma un Consorcio de Universidades de adhesión voluntaria, creando el Consorcio SIU y sustentando la financiación del equipo de trabajo mediante el aporte de 46 Universidades nacionales. En el año 2013, el financiamiento y la gestión del Consorcio SIU pasó a quedar bajo la dependencia directa del CIN [104], con la consiguiente consolidación de su estructura y planta del personal a cargo.

Además de proveer soluciones informáticas a las universidades nacionales, el SIU lo hace también a diversos órganos como el CIN y la Secretaría de Políticas Universitarias (SPU), con el objetivo de que éstas cuenten con la

información adecuada del ámbito educativo nacional cuando así lo requieran. Todos estos sistemas interactúan entre sí: la información provista por uno, es la que resulta requerida por el otro para su consumo.

Más allá de las instituciones que conforman el sistema universitario nacional argentino y sus órganos de control, los sistemas provistos por el SIU son utilizados en la actualidad por diversos organismos nacionales, provinciales y municipales [105]-[106].

La Figura 22 muestra una infografía con un resumen de la ejecución de proyectos SIU, en números, publicado en el sitio web del SIU¹⁵. Mientras tanto, en la Figura 23 se muestra otra infografía con los diferentes indicadores de la ejecución de los proyectos reportados dentro del ámbito del SIU [4].

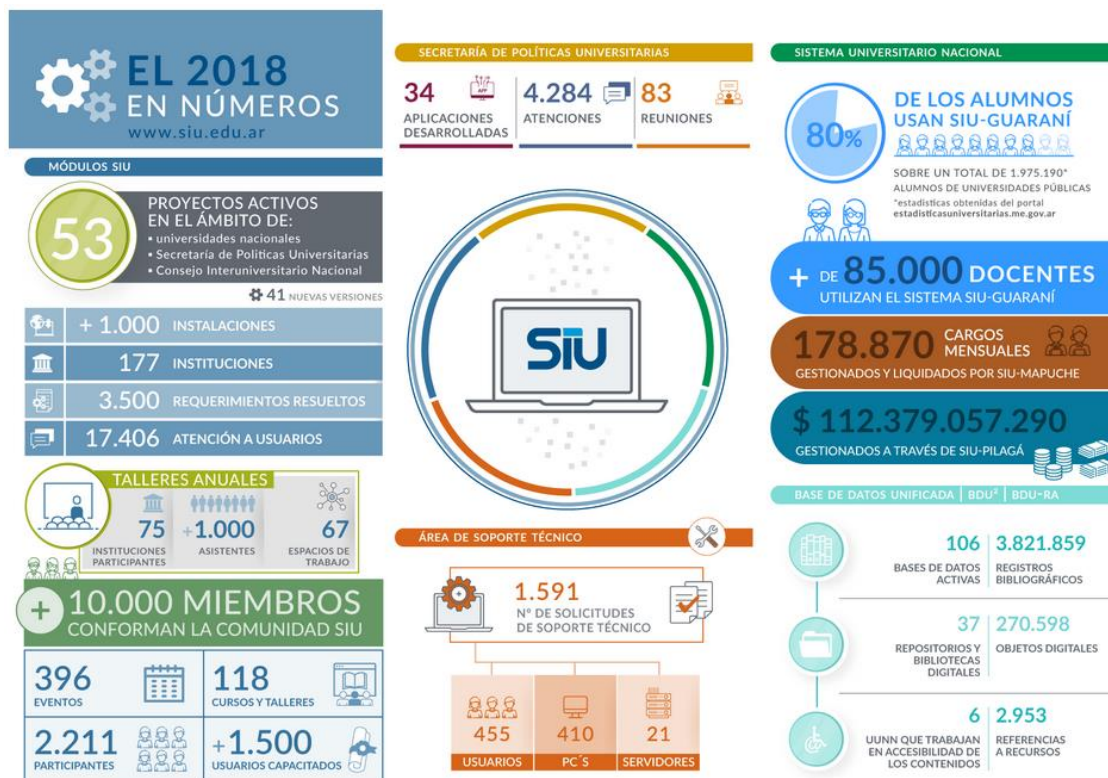


Figura 22 Resumen de la ejecución de proyectos SIU, en números [2]

¹⁵ <https://www.siu.edu.ar/nuestra-gestion-en-numeros/>

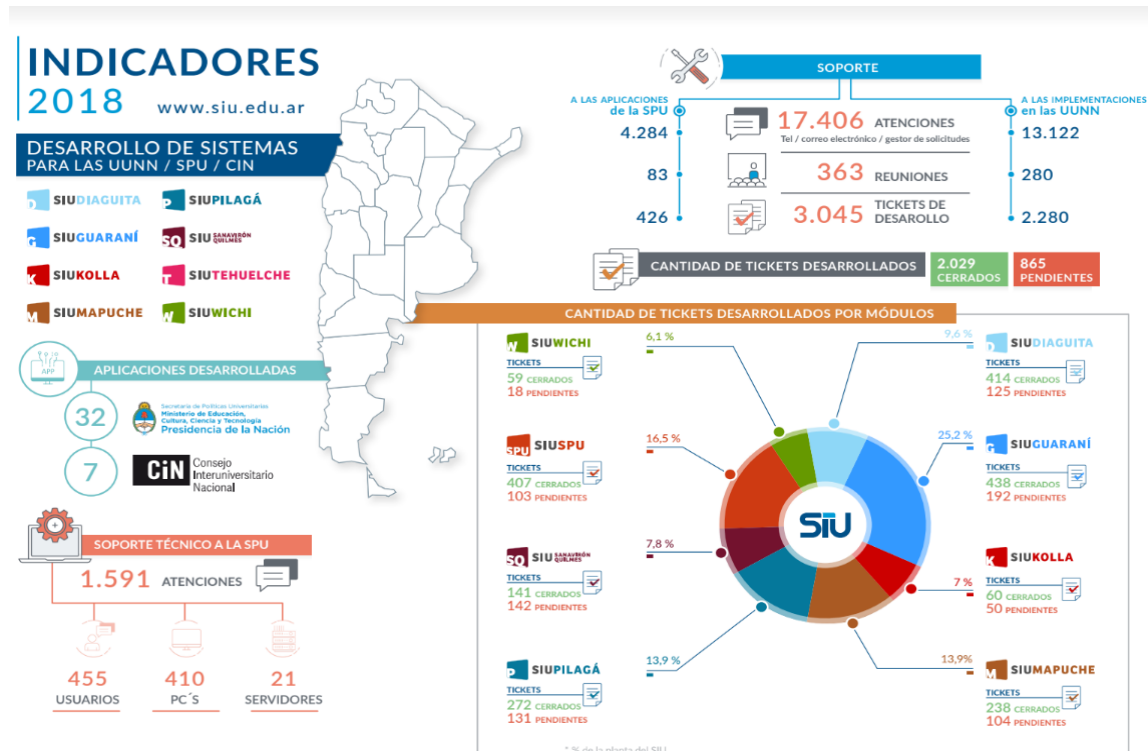


Figura 23 Indicadores de la ejecución del SIU [2]

2.3.1.1 Objetivos y Actividades

El SIU, como se lo conoce comúnmente, tiene como fin dotar a cada una de las instituciones universitarias que componen el sistema universitario argentino de soluciones informáticas integrales que colaboren a mejorar la gestión y la calidad de los datos que día a día se producen en su ámbito.

Desde el punto de vista técnico, un objetivo del SIU es impulsar la implementación, capacitación y soporte continuo a las soluciones software que ofrece. Para ello se enfoca en gran medida en desarrollar sistemas informáticos para apoyar la gestión y la toma de decisiones, junto con el análisis institucional pertinente.

Desde el punto de vista de los recursos humanos y los procesos, su objetivo consiste en desarrollar las capacidades de los distintos actores para mejorar la gestión de la información, capacitar al personal administrativo y jerárquico. Tiene responsabilidad en mostrar cómo la tecnología puede servir para abordar la mejora de los procesos y la toma de decisiones.

El desarrollo de sistemas de gestión para las universidades, como uno de

sus objetivos, busca sostener la construcción de un sistema de información para la educación superior, en el que estén involucradas y vinculadas todas las áreas que la comprenden. Para lograrlo, las actividades que lleva adelante el SIU incluyen [106]:

- El desarrollo, la implantación y el mantenimiento de productos de software de gestión.
- La permanente participación de los actores a través de los comités de usuarios, foros de discusión, reuniones, talleres, desarrollos cooperativos, etc., así como la definición de estándares tecnológicos y metodológicos para la construcción de software.
- La capacitación específica para la administración y el manejo del equipamiento y los sistemas.
- La puesta en común de los recursos de gestión actualmente disponibles en las universidades.
- El asesoramiento, monitoreo y permanente evaluación de las actividades y los resultados para la posterior retroalimentación, como base para asegurar la mejora continua.
- El estudio del mercado e investigación en cuanto a las nuevas tecnologías de la información.
- Apertura a la prestación de cualquier otro servicio que sea requerido y en el que el SIU aporte valores diferenciales respecto a otras consultoras u organismos del Estado.

2.3.1.2 Organización

El SIU en más de veinte años de gestión [1] impulsa la transformación de varios aspectos de la gestión de las instituciones universitarias; por una parte, los sistemas SIU informatizan procesos y brindan servicios a través de la tecnología; por otra parte, permiten disponer de información, que, siendo debidamente analizada, se transforma en un importante insumo para conocer en profundidad cada institución. Según Gurmendi [107], “el SIU entiende que la única forma de que un proyecto de información sea exitoso es que todos los actores involucrados en el mismo sean parte del proceso, puedan así aportar identificando necesidades

y colaborar en la decisión del rumbo del mismo”. Para lograr esto, adopta una modalidad de trabajo colaborativo y realiza una gestión basada en proyectos.

2.3.1.2.1 *Modalidad de trabajo colaborativo*

De acuerdo a Gurmendi [107], el SIU adopta y promueve una modalidad de trabajo colaborativo en red, lo que proporciona una visión consolidada y potencia los esfuerzos en áreas administrativas, gerenciales y técnicas. Según Falivene *et al* [3], esto supone una participación activa de todos los actores, logrando con ello abrir la barrera para la adopción de nuevos procesos, o la transición hacia las mejoras sobre los mismos.

El SIU propicia la participación de las Universidades, intercambiando experiencias, incrementando la eficiencia en la utilización de los recursos, entretejiendo la red del trabajo conjunto y desarrollando los acuerdos necesarios para el mejor aprovechamiento de las disponibilidades en el conjunto del sistema universitario. Se considera que esta filosofía de trabajo ha resultado pertinente para promover los aprendizajes de las Comunidades de Práctica que operan en ámbito del SIU [3]. Inclusive se trata de ir un poco más allá e impulsar esta forma de participación al interior de las instituciones. Todo esto se basa en una permanente interacción con los distintos actores de las universidades a través de los Comités de Usuarios y los Comités de Desarrollo. Estos grupos constituyen lo que se denomina como Comunidades de Práctica [108], [107], [109], [110], constituidos para lograr la innovación tecnológica colectiva, el fortalecimiento institucional y el apropiamiento de la tecnología.

- Los *Comités de Desarrolladores*, consisten en comunidades de desarrollo de software compuestas por expertos informáticos de alguna universidad y el equipo de trabajo del SIU. En ellas se definen los lineamientos y el alcance de las aplicaciones a desarrollar.
- Los *Comités de Usuarios*, donde se vinculan, en entornos estables, los usuarios finales de las instituciones y el equipo de trabajo del SIU (cada comité tiene usuarios específicos). El objetivo es plantear los problemas propios de la implementación de sistemas, plantear nuevos requerimientos que permitan mejorar las herramientas en forma colaborativa.

De acuerdo a Gurmendi y Williams [110], el SIU, a través de la conformación de Comunidades de Práctica, logra:

- a. introducir exitosamente una serie de sistemas informáticos
- b. cambiar la modalidad de trabajo con las UUNN, posibilitando que personas que trabajaban cada una en su área (como si tratase de compartimentos estancos), comiencen a interactuar no solo a nivel institución sino con sus pares de otras instituciones.
- c. promover la construcción de un sentido de pertenencia, buscando que los actores se apropien del proyecto y valoren el trabajo de su par.

2.3.1.2.2 *Gestión orientada a proyectos*

Algunos sistemas se gestaron desde su comienzo en el SIU y en otros casos se adoptaron proyectos ya existentes y se les aplicó la misma metodología de trabajo. La base de esta metodología es siempre el trabajo en comunidad o trabajo colaborativo. En torno a cada proyecto se conforma un equipo de trabajo compuesto por integrantes del SIU, el cual tendrá una permanente interacción con los distintos actores de las universidades a través de los Comités de Usuarios y los Comités de Desarrollo [107].

Estos equipos de trabajo gozan de cierta independencia unos de otros, esto es, operan de manera independiente. No obstante, se llevan a cabo esfuerzos de integración tales como la plataforma integradora de servicios SIU-Araí [5] o el desarrollo de librerías y herramientas comunes, como un framework propio denominado SIU-Toba¹⁶ [111].

Los sistemas SIU se dividen según sus destinatarios en dos categorías: sistemas que están destinados al uso interno de la universidad; sistemas destinados para uso por parte de entidades de seguimiento y control de las universidades, tales como la SPU y/o el CIN. Estos dos tipos de sistemas interactúan entre sí ya que la información provista por uno, es la que resulta requerida por los otros. En la Figura 24 se muestra en conceptos generales esta

16 SIU-Toba: es la plataforma estándar de desarrollo de los sistemas SIU. Permite la homogeneización de sus proyectos. En el año 2007, esta herramienta fue declarada software libre a través de la Resolución Ministerial No 823/2007, convirtiéndose así en el primer software de código abierto desarrollado por el Estado argentino.

interacción y los actores de los principales sistemas en el sistema universitario argentino ¹⁷.



Figura 24 Flujo de información entre UUNN y organismos [2]

También es posible clasificar los sistemas según el perfil del usuario destinatario en dos categorías. Por un lado, aquellos sistemas del nivel operacional o transaccionales, en el que se registran los movimientos del día a día; por otro lado, aquellos sistemas del nivel gerencial o de toma de decisiones, que acumulan la información producida por todos los sistemas de gestión. Según Menéndez y Gurmendi [112], este esquema permite analizar los datos generando información al servicio de las autoridades de la universidad y de los diferentes órganos que lo requieren. La Figura 25 muestra la ubicación en la Jerarquía o Pirámide del Conocimiento de los principales módulos o sistemas en uso en el Sistema Universitario Argentino.

¹⁷ <https://www.siu.edu.ar/como-obtengo-los-sistemas/>

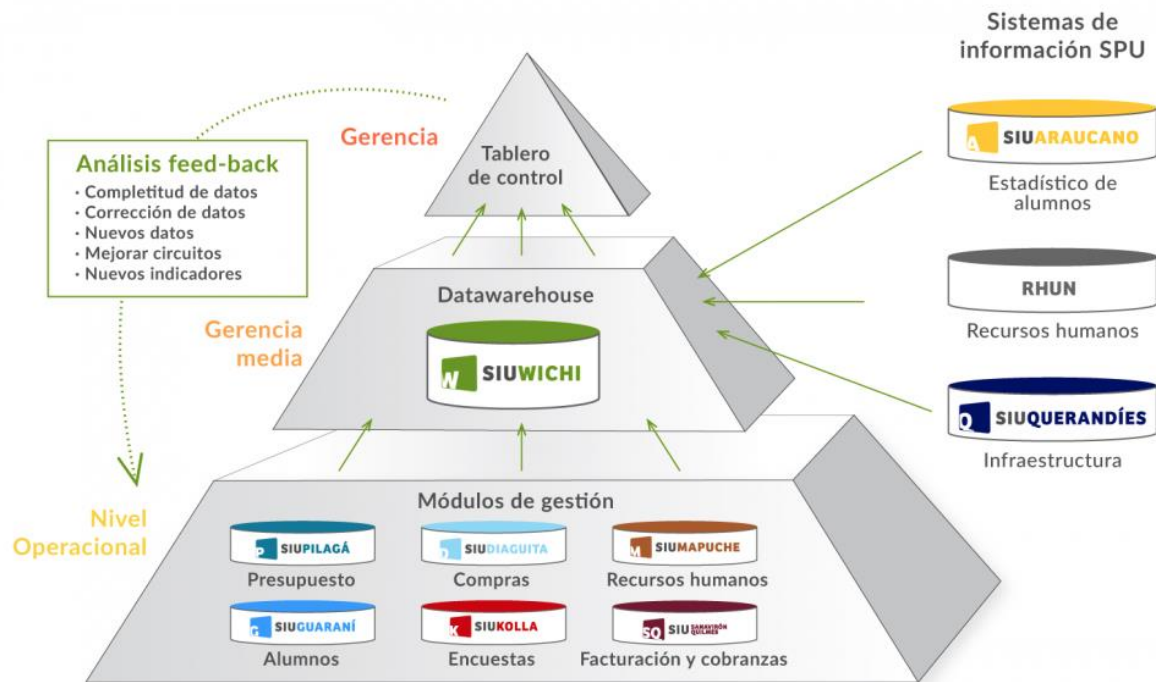


Figura 25 Tipos de proyectos SIU de acuerdo a la pirámide del conocimiento [2]

El SIU cuenta con 53 proyectos de software activos contabilizados al año 2018 [4], todos de diversos tamaños y en constante actividad (desde nuevas funcionalidades, corrección de errores, soporte, mantenimiento a la instalación y la actualización de los mismos).

2.3.2 Cómo desarrolla el Software

Esta sección describe cómo están conformados los equipos de trabajo dentro del SIU. A continuación, se describe cómo mediante la metodología de trabajo adoptada [113], el SIU gerencia el desarrollo del software con el posterior soporte técnico y mantenimiento.

2.3.2.1 Conformación de los equipos de trabajo

Los equipos de trabajo se conforman en torno a un proyecto en particular, según se detalla en la sección 2.3.1.2.2 Gestión orientada a proyectos. Cada equipo de trabajo está definido en base a un conjunto de roles y competencias, según cómo entiende el SIU en un proyecto como SIU-Guaraní [114]:

- El **Rol** es la descripción general de las tareas que debe desempeñar una persona para ejecutar en forma adecuada las funciones que tiene asignadas dentro de una estructura específica. Una persona puede llevar adelante más de un rol y, del mismo modo, un rol puede ser ocupado por más de una persona.
- Las **Competencias** son los conocimientos y habilidades necesarios para poder desempeñar las tareas descritas en el rol; adquiridos a partir de educación formal o experiencia aplicada. Existen entonces competencias que son transversales a todos los integrantes de un equipo: capacidad de trabajo en equipo; capacidad de análisis, aprendizaje y adaptación; capacidad de comunicación y capacidad para administrar tiempos.

No existe un esquema que se adapte a todos los proyectos, pero sí es común la presencia de perfiles específicos según el tipo:

- de **coordinación**, son los roles que tienen a cargo la gestión del proyecto y/o equipos de trabajo de un mismo perfil.
- el **funcional**, es un perfil de analista de software, muy empapado en el modelo de negocio del proyecto y generalmente el rol que tiene contacto directo con los usuarios finales.
- el **técnico**, es un perfil de arquitecto y/o desarrollador de software, generalmente un rol que construye el producto software.

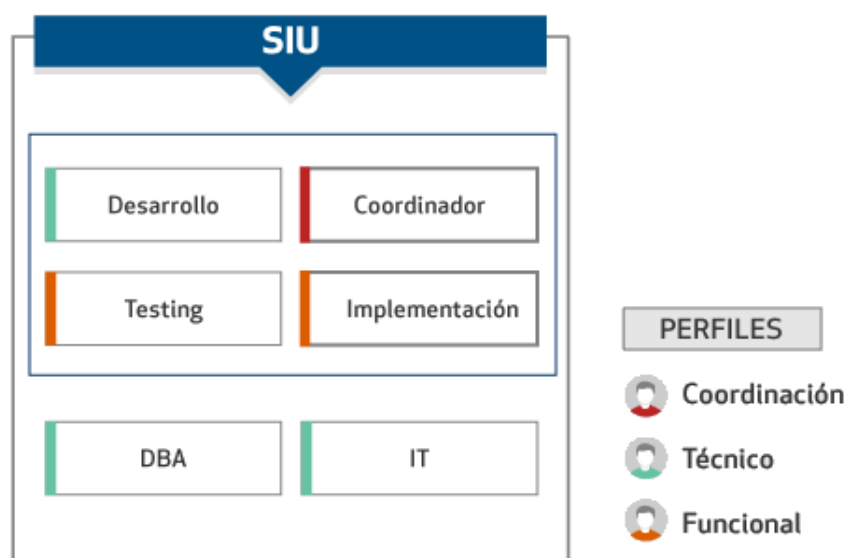


Figura 26 Agrupación de roles por perfil para equipos de trabajo del SIU [2]

El tamaño de los equipos de trabajo puede variar de acuerdo al tipo y tamaño del proyecto, así como su alcance. La Figura 26 muestra una organización de roles por grupo de perfiles, activo en el equipo de trabajo para el proyecto SIU-Guaraní [114], entre otros.

En resumen, un equipo de trabajo está conformado por tres tipos de perfiles (coordinación, funcional y técnico). Pueden existir múltiples roles (ya sea de desarrollo, pruebas, etc.). A continuación, se describen cada uno de ellos:

- **Desarrollo:** Es un perfil de tipo “técnico” el que realiza el diseño, implementación y las pruebas en un lenguaje de programación del producto software, a partir de las especificaciones.
- **Testing:** Es un perfil de tipo “funcional”, encargado de realizar las pruebas al producto software buscando el cumplimiento en base a las especificaciones.
- **Implementador:** Es un perfil de tipo “funcional”, encargado de realizar la captura, definición y análisis de los circuitos y procesos relacionados al producto software y plasmarlo en especificaciones. También está en sus funciones evaluar, coordinar, asesorar y capacitar a las UUNN en la implementación y mantenimiento de los productos software.
- **Coordinador:** Es un perfil de tipo “coordinador”, el encargado de coordinar el equipo de trabajo de perfil “funcional” y “técnico”. Si el equipo de trabajo cuenta con múltiples integrantes en roles de desarrollo y/o implementación, es posible que exista también un coordinador por cada tipo de perfil.
- **DBA/IT:** Es un perfil de tipo “técnico”, normalmente asesorando sobre cuestiones de motores de bases de datos y/o temática relacionada con la infraestructura tecnológica.

2.3.2.2 Proceso para Desarrollo de Software

Los proyectos para desarrollo de software que lleva adelante el SIU se caracterizan por estar en constante evolución, esto es, son proyectos activos [115], que operan en producción y reciben nuevas versiones (con correcciones, mejoras y/o nuevas funcionalidades) como parte de su ciclo de vida. Los plazos en los que estos productos software ven publicada una nueva versión varían desde un mes a un par de meses. Por lo tanto, y en retrospectiva, el ciclo de desarrollo resulta “reducido” si se lo compara entre un proceso de desarrollo ágil y

uno tradicional [116].

El proceso de desarrollo adaptado por el SIU [113] está basado en las metodologías tradicionales, pero teniendo algunas consideraciones e influencias de la adopción de las *Comunidades de Práctica*, según se describe en la sección 2.3.1.2.1 Modalidad de trabajo colaborativo. El principal cambio es la incorporación de una figura externa al SIU, el usuario, que siendo un integrante activo en el *Comité de Usuarios*, ayuda a definir y evaluar la aceptación de un requerimiento. En la Figura 27 se observa las fases de dicho proceso.

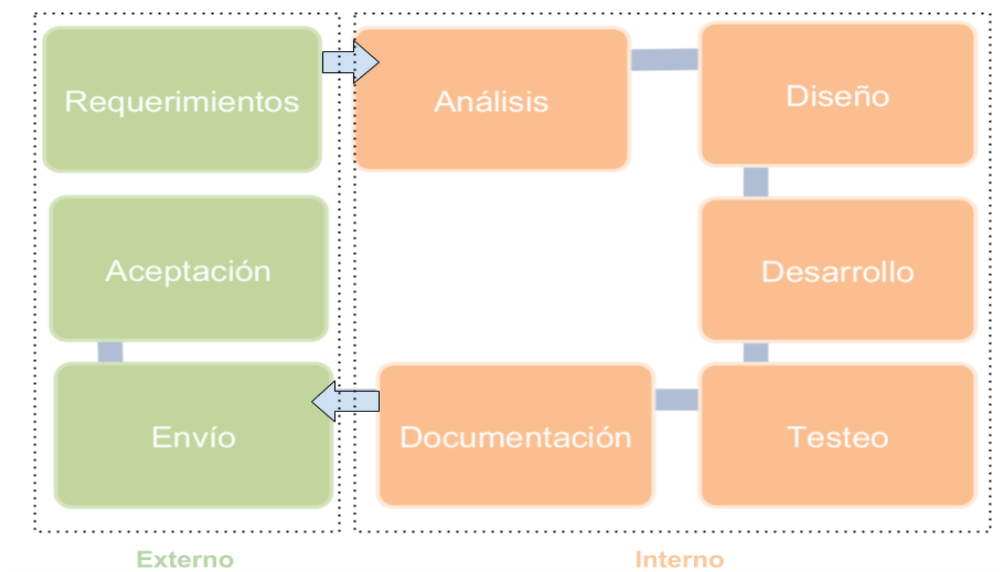


Figura 27 Metodología de desarrollo de software del SIU

A continuación, se detallan de forma breve y concisa las fases del ciclo de desarrollo de software que lleva adelante el SIU con sus equipos de trabajo:

- d. **Requerimientos:** En esta fase, usuarios activos dentro de un Comité de Usuarios establecen la necesidad de un requerimiento. Dichos usuarios pueden ser miembros de alguna UUNN, como pueden ser también los integrantes del equipo de trabajo del SIU. Su misión es definir de forma consensuada las necesidades de los circuitos académicos y administrativos que deberá soportar el producto software.
- e. **Análisis:** En esta fase, un rol funcional (coordinador, implementador) del equipo de trabajo del SIU, con conocimiento detallado de los circuitos

- académicos y administrativos de una UUNN, elabora y/o actualiza una definición de requerimiento de software. En esta etapa se prioriza el requerimiento y se define en qué versión del producto software será incluido. Este requerimiento surge habitualmente del trabajo de los integrantes del *Comité de Usuarios*.
- f. **Diseño:** En esta fase, tanto roles funcionales (coordinador, implementador) como técnicos (desarrollador, DBA, IT) del equipo de trabajo del SIU establecen las pautas de diseño para un requerimiento, elaborando las especificaciones para su implementación. Se tendrá en cuenta la naturaleza tecnológica y su integración e impacto dentro del producto software.
 - g. **Desarrollo:** En esta fase, un rol técnico (desarrollador) realiza la implementación en un lenguaje de programación a partir de las especificaciones técnicas.
 - h. **Testeo:** En esta fase, un rol funcional (tester) realiza diferentes pruebas sobre el requerimiento, su integración al producto software, así como su impacto en los circuitos. Este testeo si es exitoso se toma como de aceptación de la implementación inicial del requerimiento.
 - i. **Documentación:** En esta fase, un rol funcional (tester, implementador) documenta en detalle la operatoria del circuito académico/administrativo que ha sido implementado y/o adaptado. Una vez publicada la versión del producto software, esta documentación estará disponible en la versión en línea [117].
 - j. **Envío:** En esta fase, un rol funcional (coordinador, implementador) indica que el requerimiento ha sido completado.
 - k. **Aceptación:** En esta fase, usuarios activos dentro de un *Comité de Usuarios* establecen la aceptación de un requerimiento. Esto se realiza por medio de diferentes formas: el usuario puede acceder a una versión temprana del producto software con dicho cambio o puede evaluarlo ya en una versión publicada. Reuniones del *Comité de Usuarios* facilitan esta interacción.

2.3.2.2.1 Flujos de apoyo al proceso de desarrollo de software

En la sección anterior, se describen las fases del proceso de Desarrollo de Software, agrupándolas si son de interacción **externo** o **interno**, respecto al SIU. Para permitir el trabajo en las fases de tipo **externa**, el SIU desarrolló un servicio denominado Gestor de Solicitudes (GDS) [113]. Este servicio da soporte al conjunto de actividades externas y permite una interacción en línea con los integrantes del Comité de Usuarios en relación a los requerimientos que se establecieron. Un diagrama de transición de estados para las actividades realizadas con GDS se muestra en la Figura 28.

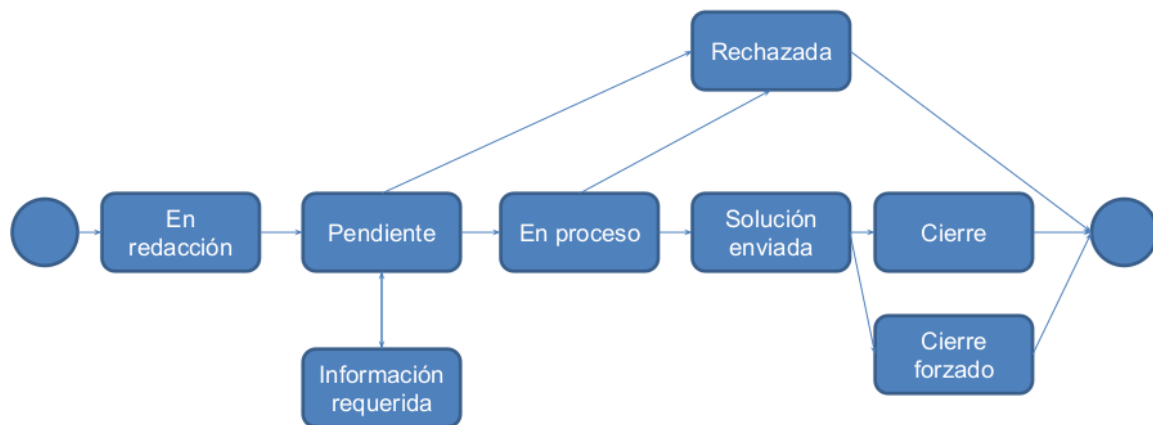


Figura 28 Estados de los tickets en el sistema de Gestión de Solicitudes [113]

Por el contrario, para apoyar las actividades y el trabajo en las fases de tipo **interna**, el SIU adopta productos estándar para la gestión de proyectos, tales como Trac¹⁸, Redmine¹⁹ y Gitlab²⁰. Esto es debido a su flexibilidad e integración con otros entornos y herramientas de trabajo. Un diagrama de transición de estados para las actividades que se realizan en un proyecto como SIU-Diaguita [118], e implementadas con dichas herramientas, se muestra en la Figura 29.

18 Trac, <https://trac.edgewall.org/>

19 Redmine, <https://redmine.org/>

20 Gitlab, <https://about.gitlab.org/>

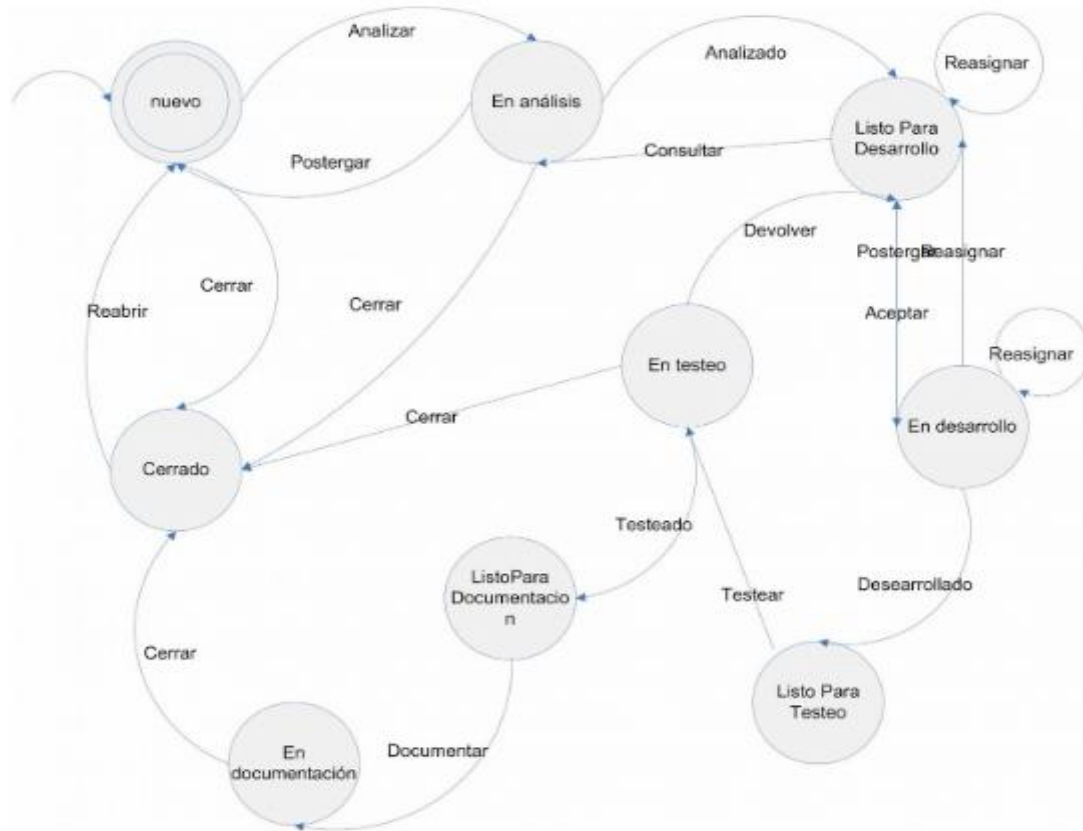


Figura 29 Transición de estados en la gestión del proyecto SIU-Diaquita [118]

2.3.2.2.2 Publicación de nuevas versiones de los proyectos

El SIU se organiza en torno a proyectos, tal cual describe la sección 2.3.1.2.2, es en torno a cada uno de estos proyectos que los equipos de desarrollo construyen en forma continua las aplicaciones. Estas aplicaciones se van liberando como nuevas versiones de producto software, al ritmo que crean conveniente los proyectos que las gestionan. El versionado de dichos proyectos sigue el esquema de *Versionado Semántico* [119], que propone un formato de versión del tipo X.Y.Z (mayor.menor.corrección). En esencia, se trata de seguir la regla de publicación de versiones de acuerdo a los tipos de cambios que esta ofrece, según estos criterios:

- ante un cambio mayor en el sistema, rompiendo compatibilidad hacia atrás, se debe incrementar el primer dígito de la versión.
- ante un cambio menor de la aplicación, incremental, es posible incrementar el dígito intermedio.
- ante una corrección de error, es posible incrementar sólo el dígito final.

Para establecer el ritmo de publicación de software que lleva a cabo el SIU, se realiza un análisis de las versiones publicadas en nueve proyectos en particular: SIU-SQ, SIU-Toba, SIU-Pilaga, SIU-Araí, SIU-Mapuche, SIU-Huarpe, SIU-Diaguita, SIU-Guaraní y SIU-Wichi respectivamente. El análisis toma datos de publicaciones comprendidos entre el 01/01/2015 y el 31/12/2019 inclusive. Los datos recopilados se muestran en la Tabla 5.

Tabla 5: Datos de análisis sobre la publicación de versiones de los proyectos

Proyecto/Aplicación	Tiempo medio entre publicación de versiones	Tiempo medio geométrico	Total versiones	Versiones bugfix
SIU-SQ	21,51	11,59	57	52
SIU-Toba	24,75	11,52	78	73
SIU-Pilaga	37,76	17,51	49	42
SIU-Arai	43,38	18,98	35	29
SIU-Mapuche	46,30	30,76	33	21
SIU-Huarpe	61,4	21,34	15	8
SIU-Diaguita	84	61,40	20	14
SIU-Guarani	99,7	88,60	20	12
SIU-Wichi	127	98,92	13	2

En la Figura 30, se puede observar el ritmo de publicación de versiones por proyecto, contrastando la cantidad de versiones publicadas por sobre la cantidad de versiones consideradas de *corrección de errores* o *bugfix*. No se evalúa si un proyecto cumple el versionado semántico en forma estricta.

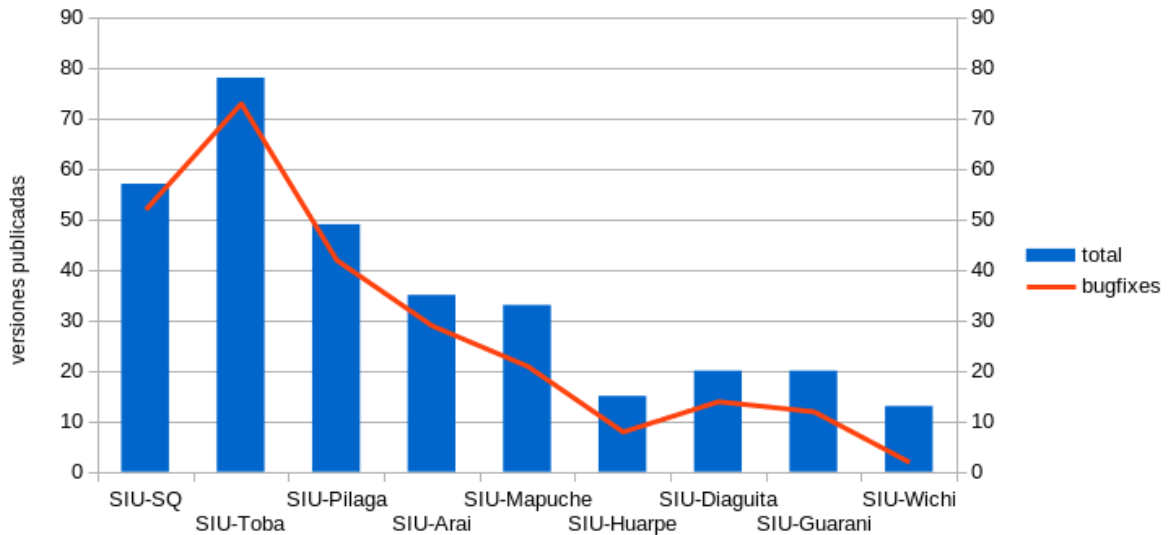


Figura 30 Publicación de versiones, por proyecto

La Figura 31 permite observar el ritmo de publicación por proyecto, es decir el tiempo promedio que tiene entre cada publicación de versión un proyecto dado. Se muestra los datos de tiempo promedio, en contraste con el promedio geométrico. La razón es simple, evitar que los valores extremos puedan ocultar un ritmo de publicación más sensible (como en el caso del proyecto SIU-Huarpe, las versiones se suceden en promedio cada 2 meses, pero reacciona en promedio liberando versiones cada 20 días).

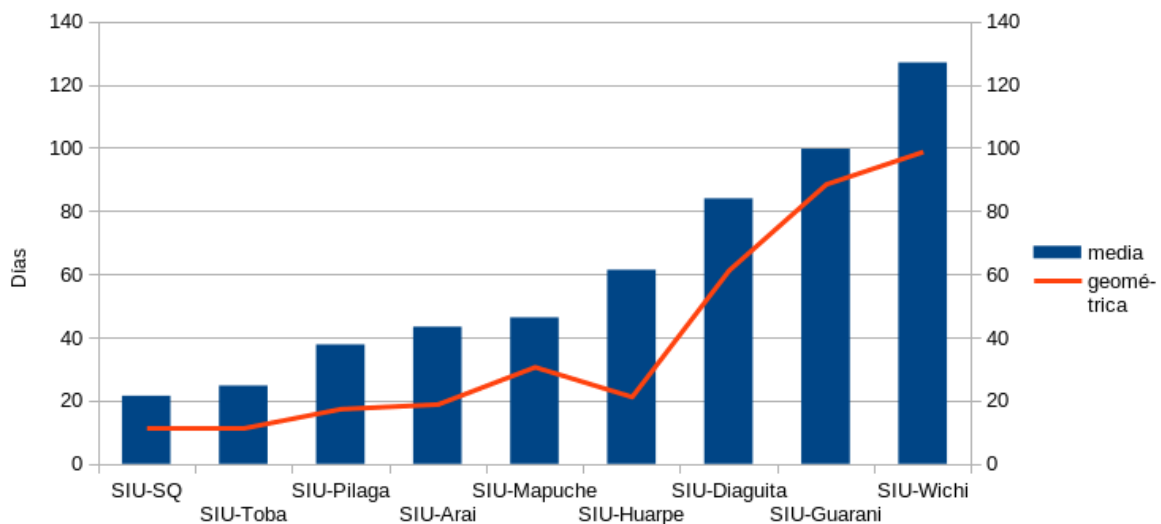


Figura 31 Tiempo entre la publicación de cada versión, por proyecto

2.3.3 Cómo realiza las Pruebas de Software

Como se mencionó previamente, el SIU tiene en desarrollo una gran variedad de proyectos [4], abarcando un amplio espectro actividades relacionadas al sistema universitario argentino (según se detalla en la sección 2.3.1.2.2 Gestión orientada a proyectos). Es un factor a considerar la naturaleza e idiosincrasia de estos equipos de trabajo en torno a los proyectos: al presentar cierta independencia unos de otros; el momento en que adoptan una decisión tecnológica y/o su transición; el impacto de un cambio en las versiones en producción, entre otros motivos vinculados a la parte funcional de los mismos.

A continuación, se describen las **Pruebas de Software** que llevan a cabo en mayor o menor medida todos los proyectos del SIU, para el aseguramiento de la calidad de sus productos software.

2.3.3.1 Pruebas unitarias

En el SIU los equipos de trabajo están adoptando paulatinamente las prácticas de realizar pruebas unitarias. Estas prácticas, por ejemplo, se están implementando con herramientas como PHPUnit²¹ y SimpleTest²², populares con el lenguaje de programación PHP.

Las pruebas unitarias son implementadas por roles de perfil técnico, como un desarrollador, normalmente al momento de introducir una característica nueva o una corrección sobre alguna existente. Las herramientas que utilizan disponen de un comando que produce un reporte de la ejecución de las pruebas. Estas pruebas son ejecutadas manualmente, cuando el desarrollador lo crea oportuno.

2.3.3.2 Pruebas de interfaz de usuario

Los proyectos software que lleva adelante el SIU, son casi en su totalidad del tipo aplicaciones web²³. Su uso se da mediante un navegador web (tales como Firefox, Google Chrome) y por lo tanto el lenguaje en el que está construida la

21 PHPUnit, <https://phpunit.de>

22 SimpleTest, <http://simpletest.org>

23 Las aplicaciones web son herramientas que los usuarios pueden utilizar accediendo a un servidor web a través de internet o de una intranet, mediante un navegador web.

Interfaz Gráfica de Usuario (del inglés Graphical User Interface, GUI) es una combinación de HTML, JavaScript y CSS.

Los equipos de trabajo en el SIU realizan pruebas sobre la GUI de diferentes formas. Una de ellas es mediante una herramienta llamada Selenium²⁴, que permite automatizar este tipo de pruebas. Una de sus funcionalidades es la de grabar la interacción del usuario con el navegador en la aplicación web, para así poder reproducirla en otro momento. Es una manera simple y efectiva de construir casos de pruebas que no requieran complejidad. Es utilizado por roles de perfil técnico como un desarrollador, de forma manual y al momento de programar una característica y/o evaluar su integración en la aplicación web.

Otra forma que los equipos del SIU realiza las pruebas de GUI es la que realizan los roles de perfil funcional, como los implementadores, que evalúan si el requerimiento fue implementado de forma adecuada “haciendo uso de la aplicación web”. Con ello se está realizando una prueba de aceptación sobre la GUI y sus componentes. Estas pruebas son realizadas de forma manual, antes de dar por finalizado el ciclo de desarrollo de una característica.

2.3.3.3 Pruebas de API

La Interfaz de Programación de Aplicaciones (del inglés Application Programming Interface, API) es el mecanismo de interconexión entre sistemas que fué adoptado en los proyectos desarrollados por el SIU. En particular, desarrolla en sus sistemas una API tipo Rest: comúnmente conocido como servicio web y basada en intercambios mediante el protocolo HTTP²⁵. El objetivo siempre es el mismo: el consumo de servicios externos, así como la provisión de servicios a terceros.

En los proyectos software que desarrolla el SIU, se hace uso de una herramienta denominada SwaggerUI²⁶, la cual permite conectarse a las API Rest y brindar una forma sencilla para la interacción con los recursos que ésta ofrece.

24 Selenium, es un entorno de pruebas de software para aplicaciones basadas en la web. <https://selenium.dev/>

25 API tipo Web, se la conoce también como Servicios Web tipo RESTFull.

26 Swagger UI, herramienta que permite visualizar e interactuar con recursos de las API sin tener nada de la lógica de implementación realizada. <https://swagger.io/tools/swagger-ui/>

Adicionalmente, esta herramienta permite generar la documentación técnica relacionada con este tipo de API, facilitando su uso por terceras partes.

Mediante esta herramienta SwaggerUI, roles de perfil técnico como un desarrollador o IT, realizan pruebas y verifican de forma manual la operatoria de los Servicios Web que ofrece la aplicación. Si bien es un procedimiento sencillo, se trata de una operatoria ardua si se tiene servicios web con decenas de recursos y métodos que los manipulan.

Este tipo de pruebas de API es utilizado como una evaluación de tipo funcional, evaluando al sistema como una caja negra. Además, puede considerarse como pruebas de integración, debido a que la herramienta SwaggerUI, como cliente, interactúa con la API de la aplicación (que a su vez opera como un conjunto activo e integrado con múltiples servicios, bases de datos, servidor de aplicación, etc.). Incluso resulta viable para las UUNN, desde el punto de vista de la aceptación, emplear este procedimiento para verificar la API opera en conformidad en un ambiente de previo a producción y/o en producción si se diera el caso.

2.3.3.4 Pruebas de instalación

En los equipos de trabajo del SIU, tanto los roles de perfil técnico como funcional, evalúan la capacidad de instalación, actualización y configuración de parámetros del sistema en desarrollo. Esta actividad la realizan prácticamente en forma diaria, siendo una tarea manual tanto la elección de la versión, el conjunto de datos o parámetros que se configuran, así como la ejecución de los comandos involucrados para lograr tal fin.

En algunos casos, los equipos de trabajo cuentan con scripts o archivos de comandos, que construyen una versión del sistema en desarrollo con un juego de datos y configuración en particular. Suelen estar preparados para su ejecución automática y en forma diaria, como una tarea programada. Este modo de verificación es un proceso automatizado que simplemente evalúa esta funcionalidad. La peculiaridad de este enfoque reside en que no utiliza ningún entorno y/o herramienta que dé soporte alguno sobre este tipo de actividad.

2.3.3.5 Pruebas de integración

Las pruebas de integración son llevadas a cabo por los roles de perfil técnico, tanto desarrolladores como DBA e IT. Estas incluyen evaluar la forma en que el proyecto software se vincula con cada una de sus dependencias: en aplicaciones web se considera el servidor web y el servidor de base de datos (Apache²⁷ y PostgreSQL²⁸ respectivamente), servicios de almacenamiento volátil o temporal (Memcached²⁹ o Redis³⁰). Esta actividad es del tipo exploratoria, se suele evaluar en este punto y de forma manual si el producto software en desarrollo opera con las versiones soportadas de sus dependencias.

Otro tipo de integración que se busca evaluar, consiste en verificar si se consume algún servicio de otro módulo SIU y su comportamiento. Para ello, se debe desplegar un ambiente de pruebas que pueda contar con las versiones requeridas de ambos productos y mediante pruebas manuales verificar el resultado de la integración.

2.3.3.6 Pruebas de aceptación

En las pruebas de aceptación, llevadas a cabo al final de cada ciclo de desarrollo de un producto software, los equipos de trabajo del SIU tratan de involucrar a los usuarios de las UUNN. Si bien este tipo de prueba se inicia con un rol de perfil funcional evaluando el cumplimiento en la implementación del requerimiento, el paso final se produce cuando los usuarios de las UUNN pueden tener contacto con la implementación de la funcionalidad en el producto. Toda la retroalimentación que se obtiene en este punto suele impulsar en ciertas ocasiones cambios sobre los requerimientos iniciales. Es por ello que se busca introducir a los usuarios de las UUNN lo más temprano posible en dicho circuito.

2.3.3.7 Pruebas de regresión

Este tipo de pruebas es realizado por los equipos del SIU en el momento

27 Apache, <https://httpd.apache.org/>

28 PostgreSQL, <https://postgresql.org/>

29 Memcached, <https://memcached.org/>

30 Redis, <https://redis.io/>

en que se inicia las actividades para liberar una nueva versión del producto software. Normalmente las pruebas de regresión son llevadas a cabo de forma manual, por roles de perfil técnico como funcional, tanto sobre los procesos de la aplicación, la GUI como las API que este ofrece, de modo que se controle de acuerdo a requerimientos previos el estado de su funcionamiento.

2.3.3.8 Pruebas de compatibilidad

Debido a que los proyectos software que lleva adelante el SIU son en su gran mayoría del tipo aplicaciones web, como se mencionó previamente, la prueba de compatibilidad más habitual está relacionada con los navegadores web. Normalmente, los roles de perfil técnico como desarrolladores, están en constante contacto con uno o más combinaciones de navegadores al momento del desarrollo de la aplicación web. Lo mismo sucede con los roles de perfil funcional, que realizando otros tipos de pruebas manuales, evalúan la compatibilidad de los mismos.

Otro tipo de prueba de compatibilidad es la que está relacionada con los formatos en los que los sistemas exportan e importan información, para el intercambio con diferentes sistemas. Para evaluar la compatibilidad, roles de perfil funcional como un implementador, generan las entradas y/o salidas de forma manual y corroboran su correcto funcionamiento.

2.3.3.9 Pruebas de rendimiento

Los proyectos software que realiza el SIU tienen un amplio impacto en la comunidad universitaria (involucran a personal de la UUNN como a sus usuarios, los alumnos). Por ello, se realizan diferentes pruebas de carga, de tiempos de respuesta, de manejo de volumen de datos, según lo requiera cada proyecto en cuestión y buscando garantizar su operatoria.

Una forma que realiza el SIU las pruebas de carga es generando una base de datos con un juego de datos correspondiente. Luego en el momento en que requiera evaluar el rendimiento de una versión particular de la aplicación, apunta a dicha base de datos y comienza a operar los procesos. Muchas veces sucede que alguna UUNN aporta un juego de datos con el cual evaluar e identificar un

problema de rendimiento. Este tipo de prueba es realizada en forma manual tanto por roles de perfil técnico durante el desarrollo, como por los funcionales durante el periodo de publicación de la versión.

Otra forma de realizar estas pruebas es la que llevan a cabo ciertos proyectos, como SIU-Guaraní, donde requieren el uso de herramientas como Apache JMeter³¹. Con ella es posible la construcción de casos de pruebas para carga de datos, la posterior ejecución de dichos casos con un número de usuarios en simultáneo, y el posterior análisis de los resultados obtenidos.

2.3.4 Conclusiones

Lo que se ha descrito en esta sección, contempla acciones individuales, normalmente aisladas y que requieren de una intervención manual. No obstante, como lo describe Trutner [113], las pruebas de software constituyen una etapa fundamental del ciclo de desarrollo de software en el marco de la *Metodología de Trabajo del SIU*.

Si es cuestión de alcance e impacto de estas pruebas de software, es posible tomar como referencia lo que describen Gurmendi *et al* [118] mientras explican los inicios del proyecto SIU-Diaguíta: como los diferentes tipos de pruebas de software que llevaron a cabo logran hacer partícipes a diferentes actores de las instituciones que implementan un producto software ofrecido por el SIU.

Por último, dentro del SIU existe de manera naturalizada un proceso que todos los proyectos siguen para la ejecución y seguimiento de un plan de pruebas de software. Se destaca el uso de herramientas como Trac y Redmine, en las que elaboran actividades específicas a las pruebas de software, presentes en cada iteración. Con ellas y la documentación funcional referida a cada proyecto [117], los equipos de trabajo logran realizar las pruebas de software necesarias.

31 Apache JMeter, <http://jmeter.apache.org/>

3

DESARROLLO

3 **DESARROLLO**

En este capítulo, se presenta un resumen de las prácticas continuas y las pruebas del software, de importancia para el desarrollo de software. Luego se presenta un resumen de cómo el SIU actualmente lleva a cabo estas actividades de desarrollo de software, describiéndose el problema relacionado y como la solución propuesta en este TFM se enmarca en las prácticas de desarrollo continuo. Finalmente, se presenta el modelo de procesos para pruebas continuas en el SIU.

3.1 **Las Prácticas Continuas y las Pruebas de Software**

En la sección 2.1.4, en el análisis de las pruebas de software se describen las actividades que la componen, mientras que en la sección 2.1.5 se lo ubica en el contexto del desarrollo continuo de software. En ellas se plantea que las pruebas de software:

- Representan toda actividad para evaluar la calidad de un producto y para mejorarlo mediante la identificación de defectos y problemas.
- Son un componente clave en la medición de la calidad del software.
- 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
- Con el correr de los años, se comprobó que la detección de errores al final del proceso, es muy costosa en comparación con la detección de errores en etapas tempranas del desarrollo
- Las pruebas deben cumplir ciertos requerimientos para ser implementadas en los entornos continuos.

De acuerdo a lo que se ha visto en las secciones 2.1.1, 2.1.2, 2.1.3 y 2.1.4 acerca del CSD las prácticas continuas implican:

- Como disciplina, CSD impulsa la adopción de prácticas ágiles CI, CT, CD y CDE.
- Asegura la calidad de software y reduce los tiempos de lanzamiento de

nuevas versiones del producto al mercado.

- La adopción de CI como paso previo y requisito para implementar CD o CDE.
- La automatización de las pruebas unitarias, de integración y de rendimiento, presentes en scripts de construcción, para su ejecución local como en CI.
- Los desarrolladores deben integrar código con frecuencia, con ejecuciones de pruebas sólidas y teniendo como prioridad máxima solucionar errores detectados en la integración.
- La creación de un proceso de despliegue del software a producción, repetible y confiable.
- El lanzamiento de una versión del software a producción debe ser una tarea sencilla, porque el mismo ya ha atravesado por diferentes niveles de pruebas. Es necesario automatizar todas las tareas involucradas y mantener todos los artefactos requeridos en un sistema de control de versiones.
- La incorporación de las pruebas en un marco de despliegue continuo.
- La adopción de un patrón de CD, como DP, que brinda un proceso automatizado abarcando todas las etapas (desde la integración de código en el repositorio de control de versiones, hasta que llega a manos de los usuarios finales).

La sección 2.1.3.3 explica como DP es un patrón de CD, que abarca todas las etapas desde la integración de código en el repositorio de control de versiones, hasta llegar a manos del usuario. Es un paradigma para mover código desde la descarga o *check-in*³² a producción de forma controlada, teniendo como objetivos:

- Las fases iniciales realizan la construcción de la aplicación, la cual ha de ser utilizada en fases y tareas posteriores.
- Cada tarea de una fase puede producir artefactos³³, los cuales son almacenados y quedan disponibles para otras tareas.

32 check-in: acción de descargar desde un sistema de control de versiones el estado del código fuente luego de un commit o cambio.

33 Cuando se menciona artefactos, se contemplan dependencias de la aplicación, binarios, versión instalable de la aplicación, juegos de datos, o todo elemento que requiera algún tipo de procesamiento y pueda ser utilizado en el futuro.

- Detectar errores lo más temprano en el DP como sea posible, evitando ejecutar tareas posteriores que dependan de ello.
- Desplegar y/o facilitar el despliegue de la construcción inicial y permitir ejecutar pruebas sobre el resultado.
- La construcción es exitosa sólo si se dan todas las condiciones previas.

3.2 Desarrollo de Software en el SIU

A continuación, se expone un resumen de la situación actual del desarrollo de software en el SIU, para finalizar con el planteo de la solución elaborada y una definición del DP propuesto.

3.2.1 Resumen de la situación actual

En la sección 2.3 se exponen los objetivos y actividades que lleva a cabo el SIU, con el fin de desarrollar productos software. La sección 2.3.2 explica la metodología de desarrollo de software que fue adoptada. En forma complementaria, la sección 2.3.3 describe en detalle las pruebas de software que se realizan en los proyectos SIU, para asegurar la calidad de sus productos software. En resumen, el desarrollo de software en el SIU presenta características que se detallan a continuación:

- Se enfoca en desarrollar, de forma continua, productos software que entregan valor, con calidad y en tiempos razonables.
- Los proyectos que se desarrollan están en constante evolución, recibiendo nuevas funcionalidades, actualizaciones de mejoras y/o correcciones en cada versión.
- La decisión de en qué momento ir a producción (publicar una nueva versión de un producto software) es una decisión de negocio y no es condicionado por una técnica.
- Si bien no opera el software en producción, son las UUNN que realizan el despliegue del producto.
- No se realizan despliegues continuos o CDE sobre ambientes de

producción.

- Adopta ciclos cortos para el desarrollo de software, con nuevas versiones de productos en plazos que van de 20 días a cuatro meses.
- Todas las pruebas de software son ejecutadas en forma manual durante el desarrollo de la versión del producto.
- La tarea de liberar una versión nueva de un producto software requiere de un esfuerzo considerable en pruebas, siempre al final del ciclo de desarrollo.
- El tiempo de respuesta ante pedidos de requerimientos (por parte de usuarios finales) y/o corrección de errores es considerable y depende del tiempo requerido para pasar por el circuito de atención de usuarios hasta liberar una nueva versión del producto.
- El armado y selección de los artefactos que componen una versión del producto a liberar, se realiza de forma manual.
- El producto software requiere se realice ciertos pasos para estar listo para ser desplegado en una instancia (sea de desarrollo, pruebas o producción).
- El ciclo de la retroalimentación entre el equipo de desarrollo, pruebas y operaciones es considerable, es decir, que se vea reflejado en la versión del producto software.

3.2.2 Planteo de la solución

A partir de los detalles expresados en la sección previa, así como en esta última, es posible definir una serie de acciones que, implementadas en un proyecto, permitirán demostrar uno de los objetivos de este TFM: *Elaborar una metodología aplicable al desarrollo de software en el SIU, que permita la construcción de un proceso de pruebas continuas como parte integral a un proceso de desarrollo continuo de software*. A continuación, se detallan las acciones a implementar:

- a) Aplicar prácticas ágiles del desarrollo continuo de software o CSD, en particular CI, CT y CD.
- b) Adoptar la práctica de CT, requiriendo que las pruebas de software, en su mayoría, sean automatizadas.

- c) Crear un proceso de despliegue del software, que sea repetible, confiable y automatizado, que logre operar en cualquier etapa del ciclo de desarrollo.
- d) Adoptar la práctica de gestión de la configuración
- e) Adoptar la práctica de CD, con entrega de las versiones del producto software, de acuerdo a la decisión del proyecto, permitiendo el ciclo de desarrollo y publicación habitual.

3.2.3 Definición del conducto de despliegue

En la sección 2.1.3.3 se describe el conducto de despliegue o DP, solución propuesta como un enfoque de extremo a extremo para adoptar CD. Teniendo en cuenta lo detallado en la sección 2.3.2 acerca de la metodología de desarrollo empleada en los proyectos que construye el SIU, se propone un patrón de DP compuesto por las siguientes fases:

1. **Construcción:** la fase inicia el DP e incluye todas las tareas que estén relacionadas a la construcción de una versión instalable de la aplicación. El resultado de estas tareas es conocido como un artefacto y es almacenado para su posterior uso en otras fases y tareas.
2. **Pruebas:** fase que incluye las tareas específicas para realizar pruebas de software, tales como pruebas unitarias, de instalación, de integración y de aceptación inicial. Son actividades consideradas críticas.
3. **Calidad:** fase complementaria y opcional, con tareas dedicadas al análisis de calidad sobre el producto software. Se pone el foco en el mecanismo de *feedback* que ofrece.
4. **Despliegue:** esta fase está pensada para ejecutar tareas que permitan desplegar la aplicación en un ambiente de pruebas. Con ello se logra hacer disponible, en forma totalmente automatizada, una instalación de dicha versión de la aplicación. Al disponer de la versión de la aplicación desplegada, es posible realizar pruebas, tanto manuales como automáticas, sobre el mismo. Ofrece una facilidad para probar características en desarrollo.
5. **Pruebas despliegue:** fase complementaria y opcional, con tareas para realizar pruebas automatizadas, del tipo de aceptación y/o

- retrocompatibilidad sobre el producto software desplegado. Se propone pruebas de tipo *smoke* o de humo, como fundamentales.
6. **Publicación:** es la última fase del DP, incluye tareas que permiten publicar la versión de la aplicación. Se toman los artefactos generados en tareas previas, almacenados para ser identificados de forma acorde.

Todas estas fases se relacionan entre sí, conformando una especie de conducto: cada cambio de la aplicación atraviesa por distintas fases y tareas, ejecutándose hasta finalizar con éxito o fallar con algún error en alguna de ellas. La Figura 32 muestra una representación de alto nivel del DP propuesto para el desarrollo de software en el SIU. El diagrama expone dos características del DP: obligatoriedad y paralelismo de las fases. Las tareas de fases **Publicación**, **Despliegue** y **Pruebas** dependen que la anterior finalice con éxito. En cambio tareas en la fase **Calidad** así como en **Pruebas despliegue** son opcionales, es decir, si alguna falla no causa la falla del DP.

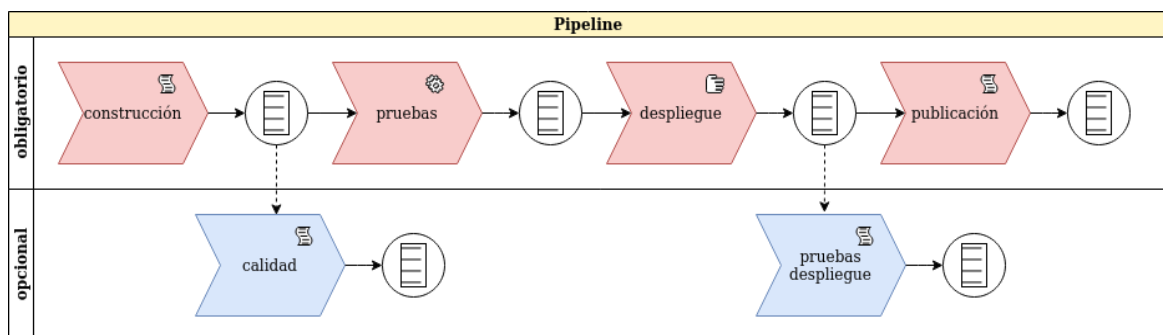


Figura 32 DP propuesto para el desarrollo de software en el SIU

3.2.3.1 Personalización del conducto de despliegue

El objetivo es lograr un DP que ofrezca fases y tareas de acuerdo a las posibilidades de trabajo de los equipos y proyectos del SIU. Esto permite modelar dos características que introducen variación en dichas tareas y fases del DP, a ejecutarse en un momento dado:

- el uso del patrón de desarrollo *branch by feature*³⁴, diseñado para facilitar

34 Branch by Feature: crear ramas por características, vinculado al desarrollo con sistemas de control de versiones tales como Git, SVN, Mercurial, entre otros.

la tarea de desarrollo en equipos y/o con trabajo en simultáneo, mientras la rama principal se mantiene estable. Esto permite crear una rama adicional (partiendo de la principal), donde se puede elaborar una característica. Esto puede suceder en paralelo con otras características. Cuando esté finalizada se procede a unir esa rama hacia la principal.

- la distribución de la aplicación para su instalación tanto en formato de código fuente como en formato contenedor, basado en tecnologías de virtualización de contenedores [122] (tales como contenedores Docker [123]). Implementar este tipo de estrategia para la operación de las aplicaciones desde el mismo momento del ciclo de desarrollo, permite potenciar la adopción de CSD y sobre todo alcanzar mejoras fundamentales planteadas en la sección 2.1.3 como el despliegue automatizado.

3.2.3.2 Modelado en lenguaje SPEM

Todas las organizaciones involucradas en el desarrollo de software necesitan establecer, gestionar y soportar el trabajo de desarrollo. El término “proceso de desarrollo de software” tiende a unificar todas las actividades y prácticas que cubren esas necesidades. Según Dominguez, *et al* [120], *“...modelar el proceso de software es una forma para mejorar el desarrollo y la calidad de las aplicaciones resultantes. De entre todos los lenguajes existentes para el modelado de procesos, aquellos basados en productos de trabajo son los más adecuados. Uno de tales lenguajes es SPEM (Software Process Engineering Metamodel)”*.

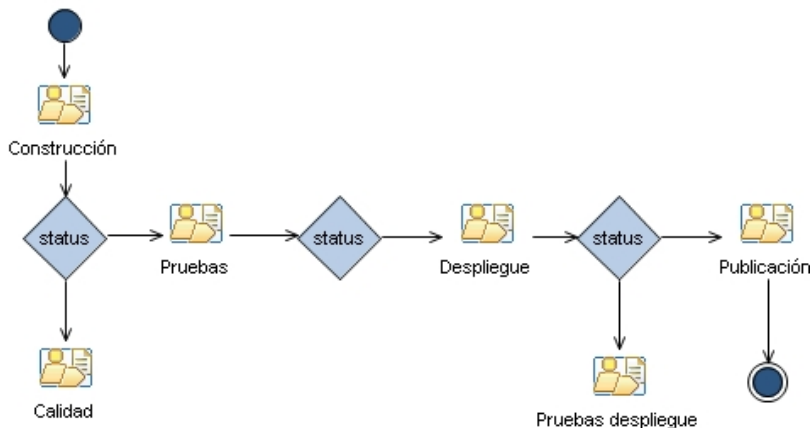
SPEM [121], en su versión 2, sirve para definir procesos de desarrollo de software y sistemas y sus componentes. Su alcance se limita a los elementos mínimos necesarios para definir dichos procesos sin añadir características específicas de un dominio o disciplina particular. No es un lenguaje de modelado de procesos en general, ya que está orientado a los procesos software.

Este TFM hace uso de SPEM para la representación del modelo de procesos para pruebas continuas, de modo que pueda ser utilizado como una referencia y la base para su aplicación futura.

3.3 Modelo de procesos para pruebas continuas en el SIU

Tomando en cuenta estas definiciones para el DP, se plantea en la sección 3.3.1 y 3.3.2 el *Modelo de procesos para pruebas continuas en el SIU*. Se propone un modelo de procesos, diseñado en el lenguaje SPEM versión 2, utilizando la herramienta EPFC según la guía de pasos propuesta por Ruiz, *et al* [124].

3.3.1 Proceso para construcción de la aplicación

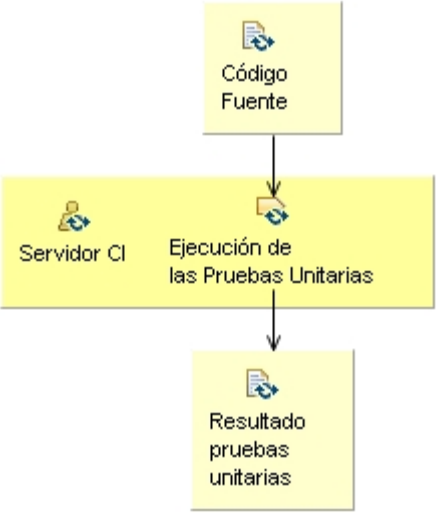
Proceso	A1. construcción código fuente
Nombre	Construcción de la aplicación a partir del código fuente
Propósito	Enfocado en evaluar en forma intensiva desde el punto de vista del código fuente. Se prueba: • cumplimiento de las reglas de negocio • calidad del código fuente • requerimientos no funcionales, tales como el despliegue manual Es un proceso considerado obligatorio para el desarrollo de cualquier tipo de proyecto.
Tipo	Obligatoria
Etapas	Obligatorias: • Construcción • Pruebas • Despliegue • Publicación Opcionales: • Calidad • Pruebas despliegue
Flujo de trabajo	 <pre> graph TD Start(()) --> Construcción[Construcción] Construcción --> Status1{status} Status1 --> Calidad[Calidad] Status1 --> Pruebas[Pruebas] Pruebas --> Status2{status} Status2 --> Despliegue[Despliegue] Status2 --> Despliegue[Despliegue] Despliegue --> Status3{status} Status3 --> PruebasDespliegue[Pruebas despliegue] Status3 --> Publicación[Publicación] Publicación --> End((())) </pre>

3.3.1.1 Etapa A1.1: construcción

Etapa	A1.1 construcción
Nombre	Construcción
Propósito	Ensamblado de los artefactos que forman parte indispensable de la aplicación. Estos artefactos: • pueden conformarse de código fuente, dependencias, binarios o ejecutables, empaquetados, documentación, entre otro material que sea necesario para distribuir con la aplicación. • son usados en primera instancia por otras tareas en estas y otras fases del conducto de despliegue. • al finalizar, pueden quedar disponibles para su consulta a posterior. Esta etapa es obligatoria y se declara como dependencia de otras etapas posteriores.
Tipo	Obligatoria
Tareas	• Construcción de las dependencias
Flujo de trabajo	 <pre> graph TD A[Código Fuente] --> B[Construcción de las dependencias] B --> C[Dependencias] </pre>

Tarea	A1.1.1 Construcción de las dependencias
Propósito	Realiza la construcción de las dependencias de la aplicación. Esta tarea puede implicar: • en sistemas tipo aplicación web, empaquetar dependencias generadas por herramientas como Composer, Yarn, Maven, etc. • en sistemas que requieran binarios, realizar compilaciones y construcción de librerías o desplegables, tales como Maven, ANT, etc. Estos artefactos generados se almacenan para su posterior utilización.
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Código fuente
Salidas	• Dependencias

3.3.1.2 Etapa A1.2: pruebas

Etapa	A1.2 pruebas
Nombre	Pruebas
Propósito	Evaluación preliminar de la aplicación mediante la ejecución de diferentes pruebas. En esta etapa, las pruebas: • buscan validar reglas de negocio o requerimientos funcionales de la aplicación. • han de ser ligeras, encapsuladas, repetibles, pequeñas y localizadas. • han de encontrar fallos de manera temprana en el conducto de despliegue, con el fin de evitar aparezcan en fases posteriores (con mayor tiempo de ejecución). La etapa de Construcción es una dependencia directa de esta etapa.
Tipo	Obligatoria
Tareas	• Ejecución de las pruebas unitarias
Flujo de trabajo	 <pre> graph TD A[Código Fuente] --> B[Ejecución de las Pruebas Unitarias] B --> C[Resultado pruebas unitarias] B --- D[Servidor CI] </pre>

Tarea	A1.2.1 Ejecución de las pruebas unitarias
Propósito	Evaluar mediante pruebas unitarias el código fuente, verificando su funcionamiento interno. Es necesario que estas pruebas unitarias: • sean ligeras, requiriendo el menor tiempo posible de ejecución. • sean autocontenidas, no requiriendo integrar con otros componentes como bases de datos.
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Código fuente
Salidas	• Resultado pruebas unitarias

3.3.1.3 Etapa A1.3: calidad

Etapa	A1.3 calidad
Nombre	Calidad
Propósito	Análisis del código fuente y sus dependencias, en búsqueda de patrones que afecten el desarrollo de la aplicación en forma negativa: • comportamiento • nivel de mantenibilidad y capacidad de crecimiento • seguridad Esta etapa es opcional, su resultado no debería detener la ejecución del conducto de despliegue en principio, pero ha de servir como retroalimentación al equipo de desarrollo acerca del estado actual del nivel de calidad del desarrollo La etapa de Construcción es una dependencia directa de esta etapa.
Tipo	Opcional
Tareas	• Evaluación mediante Análisis Estático • Evaluación de las dependencias externas
Flujo de trabajo	<pre> graph TD CF1[Código Fuente] --> E1[Evaluación mediante Análisis Estático] CF2[Código Fuente] --> E2[Evaluación de las dependencias externas] E1 --> R1[Resultado análisis estático del código] E2 --> R2[Resultado análisis de las dependencias del código] </pre>

Tarea	A1.3.1 Evaluación mediante análisis estático
Propósito	Evaluar mediante análisis estático del código fuente: • su estructura. • la aplicación de patrones de desarrollo. • malas prácticas y brechas de seguridad • errores de código • Intenta encontrar el uso de prácticas no aconsejadas, de acuerdo al lenguaje utilizado, llegando a detectar posibles errores de la aplicación en forma anticipada gracias a ello.
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Código fuente
Salidas	• Resultado análisis estático del código

Tarea	A1.3.2 Evaluación de las dependencias externas
Propósito	Evaluar mediante un análisis de <i>vulnerabilidades de seguridad bien conocidas</i>, dependencias que utiliza la aplicación. Este análisis permite conocer: • si la aplicación está utilizando alguna librería en una versión específica, la cual tiene declarada alguna debilidad en una base de datos pública, como la CVE ³⁵ (<i>Exposición y Vulnerabilidad Comunes</i>). • si existe una versión ya publicada que corrija la vulnerabilidad.
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Código fuente
Salidas	• Resultado análisis de las dependencias del código

35 CVE, <https://cve.mitre.org>

3.3.1.4 Etapa A1.4: despliegue

Etapa	A1.4 despliegue
Nombre	Despliegue
Propósito	Ejecución de los pasos necesarios para que la aplicación quede en un estado de funcionamiento adecuado. Debe ser lo más cercano a un ambiente de producción. El resultado es la aplicación disponible para realizar pruebas de integración y aceptación automáticas, así como pruebas manuales o por un ser humano. Algunas de las actividades que pueden llevarse a cabo: • realizar una instalación y/o actualización desatendida, sin intervención humana. • generación de un ambiente de datos y hacerlo disponible mediante la aplicación. • gestión de configuración y parametrización adecuada de acuerdo al ambiente de despliegue. Esta etapa es obligatoria, pues evalúa la capacidad de realizar el despliegue en los ambientes compatibles. La etapa de Construcción es una dependencia directa de esta etapa.
Tipo	Obligatoria
Tareas	• Instalación de la aplicación • Actualización de la aplicación
Flujo de trabajo	<pre> graph TD subgraph Inputs D1[Dependencias] CF1[Código Fuente] D2[Dependencias] CF2[Código Fuente] end subgraph Tasks T[Tester] I[Instalación de la Aplicación] A[Actualización de la Aplicación] end D1 --> I CF1 --> I D2 --> A CF2 --> A </pre>

Tarea	A1.4.1 Instalación de la aplicación
Propósito	Ejecuta los pasos para realizar una instalación de la aplicación desde cero. Parte de una gestión de configuración y parametrización puntual para el ambiente. Normalmente esta tarea verifica el camino habitual de una instalación limpia y la consecuente generación del conjunto de datos iniciales.
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Dependencias • Código fuente
Salidas	• Aplicación instalada

Tarea	A1.4.2 Actualización de la aplicación
Propósito	Ejecuta los pasos para realizar una actualización de la aplicación, sobre una versión previa existente. Parte de una gestión de configuración y parametrización puntual para el ambiente. Normalmente esta tarea verifica el camino habitual de una actualización y las consecuentes tareas de migración que sean necesarias.
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Dependencias • Código fuente • Aplicación en versión anterior instalada
Salidas	• Aplicación actualizada

3.3.1.5 Etapa A1.5: pruebas despliegue

Etapa	A1.5 pruebas despliegue
Nombre	Pruebas despliegue
Propósito	Evaluación a mayor profundidad y nivel de detalle sobre la aplicación ya desplegada y en funcionamiento, mediante la ejecución de diferentes pruebas. Busca validar reglas de negocio, requerimientos funcionales y no funcionales de la aplicación. Estas pruebas han de ser: • pruebas de integración rápida, tipo <i>smoke tests</i>. • pruebas de integración, tipo GUI/API/componentes • pruebas de aceptación, evaluando requerimientos funcionales • pruebas manuales, ejecutadas por un humano accediendo a la aplicación en el ambiente desplegado. Esta etapa ha de ser opcional, aunque recomendable su ejecución a medida que los casos de prueba sean sólidos y consistentes entre ejecuciones. La etapa de Despliegue es una dependencia directa de esta etapa.
Tipo	Opcional
Tareas	• Ejecución pruebas de verificación rápida • Ejecución pruebas de integración API • Ejecución pruebas de integración GUI
Flujo de trabajo	<pre> graph TD subgraph Columnas direction LR C1[Aplicación en ejecución] C2[Aplicación en ejecución] C3[Aplicación en ejecución] end C1 --> CI[Servidor CI] C2 --> CI C3 --> CI subgraph CI_Box [] direction TB CI T1[Ejecución pruebas de integración rápida] T2[Ejecución pruebas de integración API] T3[Ejecución de pruebas de integración GUI] end CI --> T1 CI --> T2 CI --> T3 T1 --> R1[Resultado pruebas integración] T2 --> R2[Resultado pruebas integración] T3 --> R3[Resultado pruebas integración] </pre>

Tarea	A1.5.1 Ejecución pruebas de integración rápida
Propósito	Evaluar mediante pruebas de integración ligeras y rápidas, diversas funciones de la aplicación. Pueden ir desde la interfaz de programación o API hasta la interfaz gráfica o GUI. Se han de tener ciertos recaudos: • las pruebas no pueden modificar datos en la aplicación ni requerir encadenar casos en un orden determinado. • lograr una cobertura mínima, sólo de casos esenciales como podría ser el ingreso del sistema y realizar un pago. • evitar casos de pruebas de gran dimensión (por tiempo y esfuerzos que insume su creación, mantenimiento y ejecución).
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Aplicación en ejecución
Salidas	• Resultados pruebas de integración rápida

Tarea	A1.5.2 Ejecución pruebas de integración API
Propósito	Evaluar mediante pruebas de integración el funcionamiento de la interfaz de programación o API. Se han de tener ciertos recaudos: • lograr una cobertura del 100% debido a que es un componente de interconexión con otros sistemas (una API es un contrato de servicios). • evitar casos de pruebas de gran dimensión (por tiempo y esfuerzos que insume su creación, mantenimiento y ejecución). • pueden modificar datos en la aplicación y requerir encadenar casos en un orden determinado.
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Aplicación en ejecución
Salidas	• Resultados pruebas de integración API

Tarea	A1.5.3 Ejecución pruebas de integración GUI
Propósito	Evaluar mediante pruebas de integración el funcionamiento de la interfaz gráfica o GUI. Se han de tener ciertos recaudos: • estresar los caminos críticos de la aplicación. • evitar casos de pruebas de gran dimensión (por tiempo y esfuerzos que insume su creación, mantenimiento y ejecución). • pueden modificar datos en la aplicación y requerir encadenar casos en un orden determinado.
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Aplicación en ejecución
Salidas	• Resultados pruebas de integración GUI

3.3.1.6 Etapa A1.6: publicación

Etapa	A1.6 publicación
Nombre	Publicación
Propósito	Hacer disponible el estado actual de la aplicación y todos los artefactos que la componen, como una versión publicada de acuerdo a un criterio semántico. Esta versión de la aplicación se considera: • contiene todo lo necesario para su operación en producción. • ha pasado por una serie de pruebas del conducto de despliegue, todas automatizadas, en forma exitosa. Esta etapa es obligatoria, pues permite liberar una versión usable de la aplicación. La etapa de Despliegue es una dependencia indirecta de esta etapa, mientras que la etapa Construcción es una dependencia directa.
Tipo	Obligatoria
Tareas	• Publicación de la aplicación • Publicación de las dependencias de la aplicación
Flujo de trabajo	<pre> graph TD subgraph Inputs D1[Dependencias Código Fuente] D2[Dependencias] end subgraph Process C[Coordinador] PA[Publicación de la Aplicación] PDA[Publicación de las dependencias de la Aplicación] end subgraph Outputs CFV[Código Fuente con versión de Aplicación] DE[Dependencias externas de la Aplicación] end D1 --> PA D2 --> PDA C --> PA C --> PDA PA --> CFV PDA --> DE </pre>

Tarea	A1.6.1 Publicación de la aplicación
Propósito	Generar uno o varios artefactos, incluyendo el código fuente de la aplicación, que representen la versión a publicar. Esta versión ha de ser posible desplegar en un ambiente de producción si así lo requiere.
Tipo	Obligatoria
Roles	• Coordinador
Entradas	• Dependencias • Código Fuente
Salidas	• Código fuente y dependencias con versión de aplicación

Tarea	A1.6.2 Publicación de las dependencias de la aplicación
Propósito	Generar uno o varios artefactos, incluyendo dependencias de aplicación, que representen la versión a publicar. Esta versión se despliega en un ambiente de producción si así lo requiere. Se proporciona como mecanismo para asegurar que en producción se utilizan exactamente las mismas dependencias con las que fue probada la aplicación.
Tipo	Obligatoria
Roles	• Coordinador
Entradas	• Dependencias
Salidas	• Dependencias externas con versión de aplicación

3.3.2 Proceso para distribución de la aplicación con Docker

Proceso	B1. distribuir aplicación docker
Nombre	Distribución de la aplicación en imagen Docker
Propósito	Enfocado en evaluar en forma intensiva desde el punto de vista de la distribución de la aplicación en imágenes Docker. Se prueba: • cumplimiento de las reglas de negocio • calidad de la aplicación y el empaquetado • requerimientos no funcionales, tales como el proceso de despliegue Es un proceso complementario al de construcción de la aplicación.
Tipo	Obligatoria
Etapas	Obligatorias: • Construcción • Pruebas • Despliegue • Publicación Opcionales: • Calidad • Pruebas despliegue
Flujo de trabajo	<pre> graph TD Start(()) --> Construcción[Construcción] Construcción --> Status1{status} Status1 --> Pruebas[Pruebas] Pruebas --> Status2{status} Status2 --> Despliegue[Despliegue] Despliegue --> Status3{status} Status3 --> Publicación[Publicación] Publicación --> End((())) Status1 --> Calidad[Calidad] Status3 --> PruebasDespliegue[Pruebas despliegue] Calidad --> Status2 PruebasDespliegue --> Status3 </pre>

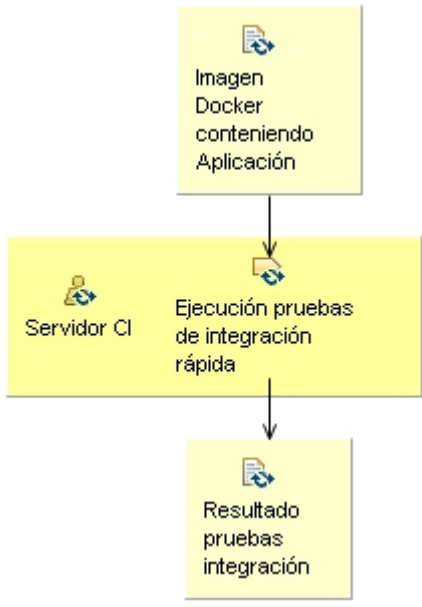
3.3.2.1 Etapa B1.1: construcción

Etapa	B1.1 construcción
Nombre	Construcción
Propósito	Ensamblado de los artefactos que forman parte indispensable de la aplicación. Estos artefactos: • pueden conformarse de código fuente, dependencias, binarios o ejecutables, empaquetados, documentación, entre otro material que sea necesario para distribuir con la aplicación. • son usados en primera instancia por otras tareas en estas y otras fases del conducto de despliegue. • al finalizar, pueden quedar disponibles para su consulta a posterior. Esta etapa es obligatoria y se declara como dependencia de otras etapas posteriores.
Tipo	Obligatoria
Tareas	• Construcción de las dependencias • Construcción de la imagen Docker
Flujo de trabajo	<pre> graph TD CF[Código Fuente] --> CI[Servidor CI] D[Dependencias] --> CI subgraph CI_Box [] direction LR CI --- CDep[Construcción de las dependencias] CI --- CIDocker[Construcción de la imagen Docker] end CDep --> D2[Dependencias] CIDocker --> ID[Imagen Docker conteniendo Aplicación] style CI_Box fill:#ffff00,stroke:#333,stroke-width:1px style D2 fill:#ffff00,stroke:#333,stroke-width:1px style ID fill:#ffff00,stroke:#333,stroke-width:1px </pre>

Tarea	B1.1.1 Construcción de las dependencias
Propósito	Realiza la construcción de las dependencias de la aplicación. Puede implicar: • en sistemas tipo aplicación web, empaquetar dependencias generadas por herramientas como Composer, Yarn, Maven, etc. • en sistemas que requieran binarios, realizar compilaciones y construcción de librerías o desplegables, tales como Maven, ANT, etc. Estos artefactos generados, se almacenan para su posterior utilización.
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Código fuente
Salidas	• Dependencias

Tarea	B1.1.2 Construcción de la imagen Docker
Propósito	Realiza la construcción de la imagen Docker conteniendo la aplicación. Puede implicar: • en sistemas tipo aplicación web, empaquetar dependencias generadas por herramientas como Composer, Yarn, Maven, etc. • en sistemas que requieran binarios, realizar compilaciones y construcción de librerías o desplegables, tales como Maven, ANT, etc. • instalación de paquetes externos que conforman la imagen Docker Este artefacto generado se almacena para su posterior utilización.
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Código fuente • Dependencias
Salidas	• Imagen Docker conteniendo aplicación

3.3.2.2 Etapa B1.2: pruebas

Etapa	B1.2 pruebas
Nombre	Pruebas
Propósito	Evaluación preliminar de la aplicación mediante la ejecución de diferentes pruebas. En esta etapa, las pruebas: • buscan validar reglas de negocio o requerimientos funcionales de la aplicación. • han de ser ligeras, encapsuladas, repetibles, pequeñas y localizadas. • han de encontrar fallos de manera temprana en el conducto de despliegue, con el fin de evitar aparezcan en fases posteriores (con mayor tiempo de ejecución). La etapa de Construcción es una dependencia directa de esta etapa.
Tipo	Obligatoria
Tareas	• Ejecución de las pruebas de integración rápida
Flujo de trabajo	 <pre> graph TD A[Imagen Docker conteniendo Aplicación] --> B[Ejecución pruebas de integración rápida] B --> C[Resultado pruebas integración] </pre> El diagrama de flujo de trabajo ilustra el proceso de ejecución de pruebas de integración rápida. Comienza con un bloque superior que indica 'Imagen Docker conteniendo Aplicación'. Una flecha descendente conduce a un bloque central que muestra 'Servidor CI' y 'Ejecución pruebas de integración rápida'. Desde este bloque, otra flecha descendente lleva a un bloque inferior que indica 'Resultado pruebas integración'.

Tarea	B1.2.1 Ejecución de las pruebas de integración rápida
Propósito	Evaluar mediante pruebas de integración ligeras y rápidas, diversas funciones de la aplicación. Pueden ir desde la interfaz de programación o API hasta la interfaz gráfica o GUI. Se han de tener ciertos recaudos: • las pruebas no pueden modificar datos en la aplicación ni requerir encadenar casos en un orden determinado. • lograr una cobertura mínima, sólo de casos esenciales como podría ser el ingreso del sistema y realizar un pago. • evitar casos de pruebas de gran dimensión (por tiempo y esfuerzos que insume su creación, mantenimiento y ejecución).
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Imagen Docker conteniendo aplicación
Salidas	• Resultado pruebas integración

3.3.2.3 Etapa B1.3: calidad

Etapa	B1.3 calidad
Nombre	Calidad
Propósito	Análisis del código fuente, así como sus posibles dependencias, en búsqueda de patrones que afecten el desarrollo de la aplicación en forma negativa: • comportamiento • nivel de mantenibilidad • capacidad de crecimiento • seguridad Esta etapa es opcional, su resultado no debería detener la ejecución del conducto de despliegue en principio, pero ha de servir como retroalimentación al equipo de desarrollo acerca del estado actual del nivel de calidad del desarrollo. La etapa de Construcción es una dependencia directa de esta etapa.
Tipo	Opcional
Tareas	• Escaneo de la imagen Docker
Flujo de trabajo	<pre> graph TD A[Imagen Docker conteniendo Aplicación] --> B[Servidor CI / Escaneo de la imagen Docker] B --> C[Resultado escaneo contenedor] </pre>

Tarea	B1.3.1 Escaneo de la imagen Docker
Propósito	Evaluar mediante un análisis de <i>vulnerabilidades de seguridad bien conocidas</i>, dependencias que utiliza la aplicación. Este análisis permite conocer: • si la aplicación está utilizando alguna librería en una versión específica, la cual tiene declarada alguna debilidad en una base de datos pública, como la CVE ³⁶ (<i>Exposición y Vulnerabilidad Comunes</i>). • si existe una versión ya publicada que corrija la vulnerabilidad.
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Imagen Docker conteniendo aplicación
Salidas	• Resultado escaneo contenedor

36 CVE, <https://cve.mitre.org>

3.3.2.4 Etapa B1.4: despliegue

Etapa	B1.4 despliegue
Nombre	Despliegue
Propósito	Ejecución de los pasos necesarios para que la aplicación quede en un estado de funcionamiento correcto, lo más cercano a un ambiente de producción. El resultado es un ambiente disponible para realizar pruebas de integración y aceptación automáticas, así como pruebas manuales. Algunas de las actividades que pueden llevarse a cabo: • realizar una instalación y/o actualización desatendida, sin intervención humana. • generación de un ambiente de datos y hacerlo disponible mediante la aplicación. • gestión de configuración y parametrización adecuada de acuerdo al ambiente de despliegue. Esta etapa es obligatoria, pues evalúa la capacidad de realizar el despliegue en los ambientes compatibles. La etapa de Construcción es una dependencia directa de esta etapa.
Tipo	Obligatoria
Tareas	• Ejecución de la imagen Docker
Flujo de trabajo	<pre> graph TD A[Imagen Docker conteniendo Aplicación] --> B[Servidor CI / Ejecución de la imagen Docker] B --> C[Resultado aplicación ejecutandose] </pre>

Tarea	B1.4.1 Ejecución de la imagen Docker
Propósito	Ejecuta los pasos para realizar el despliegue de la imagen Docker de la aplicación. Parte de una gestión de configuración y parametrización puntual para el ambiente. Normalmente esta tarea verifica el camino habitual de un despliegue de la imagen Docker desde cero y la generación del conjunto de datos iniciales.
Tipo	Obligatoria
Roles	• Servidor CI
Entradas	• Imagen Docker conteniendo aplicación
Salidas	• Resultado aplicación ejecutándose

3.3.2.5 Etapa B1.5: pruebas despliegue

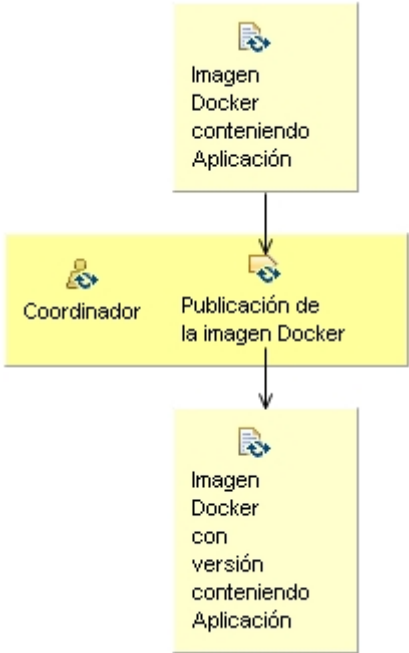
Etapa	B1.5 pruebas despliegue
Nombre	Pruebas despliegue
Propósito	Evaluación a mayor profundidad y nivel de detalle sobre la aplicación ya desplegada y en funcionamiento, mediante la ejecución de diferentes pruebas. Busca validar reglas de negocio, requerimientos funcionales y no funcionales de la aplicación. Estas pruebas han de ser: • pruebas de integración rápida, tipo <i>smoke tests</i>. • pruebas de integración, tipo GUI/API/componentes • pruebas de aceptación, evaluando requerimientos funcionales • pruebas manuales, ejecutadas por un humano accediendo a la aplicación en el ambiente desplegado. Esta etapa ha de ser opcional, aunque recomendable su ejecución a medida que los casos de prueba sean sólidos y consistentes entre ejecuciones. La etapa de Despliegue es una dependencia directa de esta etapa.
Tipo	Opcional
Tareas	• Ejecución pruebas de verificación rápida • Ejecución pruebas de integración API • Ejecución pruebas de integración GUI
Flujo de trabajo	<pre> graph TD A1[Aplicación en ejecución] --> B1[Ejecución pruebas de integración rápida] A2[Aplicación en ejecución] --> B2[Ejecución pruebas de integración API] A3[Aplicación en ejecución] --> B3[Ejecución de pruebas de integración GUI] B1 --> C1[Resultado pruebas integración] B2 --> C2[Resultado pruebas integración] B3 --> C3[Resultado pruebas integración] </pre>

Tarea	B1.5.1 Ejecución pruebas de integración rápida
Propósito	Evaluar mediante pruebas de integración ligeras y rápidas, diversas funciones de la aplicación. Pueden ir desde la interfaz de programación o API hasta la interfaz gráfica o GUI. Se han de tener ciertos recaudos: • las pruebas no pueden modificar datos en la aplicación ni requerir encadenar casos en un orden determinado. • lograr una cobertura mínima, sólo de casos esenciales como podría ser el ingreso del sistema y realizar un pago. • evitar casos de pruebas de gran dimensión (por tiempo y esfuerzos que insume su creación, mantenimiento y ejecución).
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Aplicación en ejecución
Salidas	• Resultados pruebas de integración rápida

Tarea	B1.5.2 Ejecución pruebas de integración API
Propósito	Evaluar mediante pruebas de integración el funcionamiento de la interfaz de programación o API. Se han de tener ciertos recaudos: • lograr una cobertura del 100% debido a que es un componente de interconexión con otros sistemas (una API es un contrato de servicios). • evitar casos de pruebas de gran dimensión (por tiempo y esfuerzos que insume su creación, mantenimiento y ejecución). • pueden modificar datos en la aplicación y requerir encadenar casos en un orden determinado.
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Aplicación en ejecución
Salidas	• Resultados pruebas de integración API

Tarea	B1.5.3 Ejecución pruebas de integración GUI
Propósito	Evaluar mediante pruebas de integración el funcionamiento de la interfaz gráfica o GUI. Se han de tener ciertos recaudos: • estresar los caminos críticos de la aplicación. • evitar casos de pruebas de gran dimensión (por tiempo y esfuerzos que insume su creación, mantenimiento y ejecución). • pueden modificar datos en la aplicación y requerir encadenar casos en un orden determinado.
Tipo	Opcional
Roles	• Servidor CI
Entradas	• Aplicación en ejecución
Salidas	• Resultados pruebas de integración GUI

3.3.2.6 Etapa B1.6: publicación

Etapa	B1.6 publicación
Nombre	Publicación
Propósito	Hacer disponible el estado actual de la aplicación y todos los artefactos que la componen, como una versión publicada de acuerdo a un criterio semántico. Esta versión de la aplicación se considera: • contiene todo lo necesario para su operación en producción. • ha pasado por una serie de pruebas del conducto de despliegue, todas automatizadas, en forma exitosa. Esta etapa es obligatoria, pues permite liberar una versión usable de la aplicación. La etapa de Despliegue es una dependencia indirecta de esta etapa, mientras que la etapa Construcción es una dependencia directa.
Tipo	Obligatoria
Tareas	• Publicación de la imagen Docker
Flujo de trabajo	 <pre> graph TD A[Imagen Docker conteniendo Aplicación] --> B[Coordinador Publicación de la imagen Docker] B --> C[Imagen Docker con versión conteniendo Aplicación] </pre>

Tarea	B1.6.1 Publicación de la imagen Docker
Propósito	Generar uno o varios artefactos, conteniendo la imagen Docker de la aplicación, que representen la versión a publicar. Esta versión ha de ser posible desplegar en un ambiente de producción si así lo requiere.
Tipo	Obligatoria
Roles	• Coordinador
Entradas	• Imagen Docker conteniendo aplicación
Salidas	• Imagen Docker con versión conteniendo aplicación

3.3.3 Etapas del proceso según el flujo de desarrollo

Con un sistema de control de versiones, una rama permite separar el desarrollo en paralelo de una pieza de código fuente. Se crea una rama adicional (partiendo de la principal), donde se puede elaborar la característica. El desarrollo puede suceder en paralelo con otras características. Cuando esté finalizada se procede a unir esa rama hacia la principal. Es vital que un desarrollador pueda obtener retroalimentación del DP desde el inicio del desarrollo de la característica, no solo cuando esté en condiciones de ser desplegada a un ambiente de pruebas.

En este modelo de proceso para pruebas continuas, se propone que de acuerdo al momento en el que se encuentre el desarrollo de la característica software se ejecute determinadas fases y tareas que componen el DP propuesto:

- al iniciar la característica, pues al principio del desarrollo no es requerida ninguna tarea de publicación; se busca rapidez en la retroalimentación del DP hacia el desarrollador; el desarrollador quizá ejecute múltiples veces en el mismo día el DP hasta lograr finalizar la característica.
- al finalizar la característica, el desarrollo debe estar completo o ser evaluado como si lo estuviera; al tratar de finalizar el desarrollo no es requerida ninguna tarea de publicación.
- sobre la rama principal, todas las tareas han de ser ejecutadas; ser tarea opcional permite flexibilidad a la gestión del proyecto.

3.3.3.1 Proceso para la construcción de la aplicación

Siguiendo el planteo del desarrollo de características en ramas, la Tabla 6 muestra la obligatoriedad de ejecución de las fases relacionadas al proceso para la construcción de la aplicación. Las fases pueden ser requeridas, opcionales o no requeridas, de acuerdo al momento en el que se encuentra el desarrollo de la característica software.

Tabla 6: Uso de las etapas del proceso para construcción de la aplicación

	iniciar característica	finalizar característica	rama principal
A1.1 construcción	obligatoria	obligatoria	obligatoria
A1.2 pruebas	obligatoria	obligatoria	obligatoria
A1.3 calidad	no	opcional	opcional
A1.4 despliegue	no	opcional	obligatoria
A1.5 pruebas desp.	no	opcional	opcional
A1.6 publicación	no	no	obligatoria

3.3.3.2 Proceso para la distribución de la aplicación con Docker

Siguiendo el planteo del desarrollo de características en ramas, la Tabla 7 muestra la obligatoriedad de ejecución de las fases relacionadas al proceso para la distribución de la aplicación en Docker. Las fases pueden ser requeridas, opcionales o no requeridas, de acuerdo al momento en el que se encuentra el desarrollo de la característica software.

Tabla 7: Uso de las etapas del proceso para distribución de la aplicación

	iniciar característica	finalizar característica	rama principal
B1.1 construcción	obligatoria	obligatoria	obligatoria
B1.2 pruebas	obligatoria	obligatoria	obligatoria
B1.3 calidad	no	opcional	opcional
B1.4 despliegue	no	opcional	obligatoria
B1.5 pruebas desp.	no	opcional	opcional
B1.6 publicación	no	no	obligatoria

4

VERIFICACIÓN

4 VERIFICACIÓN

En este capítulo, se presentan los pasos realizados para llevar a cabo la verificación de la solución propuesta en el capítulo 3. En la primera parte, se explica el ecosistema software que permite dar soporte a los procedimientos propuestos. Posteriormente, se detalla la preparación requerida al proyecto que actúa como caso de estudio de este TFM, el desarrollo de la plataforma SIU-Araí. Finalmente, se presenta una implementación de la solución propuesta, junto con la discusión de los resultados obtenidos.

4.1 Definición del ecosistema software

Para llevar a cabo la implementación del proceso de pruebas continuas en el SIU, y teniendo en cuenta los procedimientos desarrollados en el capítulo 3, es necesario establecer un ecosistema de herramientas software que permita dar soporte a dicha actividad. Este ecosistema debe operar integrado y cohesivo en el marco de desarrollo de software que se lleva a cabo el SIU. A continuación, se detallan una serie de herramientas software que permiten poner al SIU un paso más cerca de implementar con éxito dicho proceso.

En primera instancia, para dar un marco de trabajo a la solución propuesta en la sección 3.2.2, este TFM propone adoptar la plataforma Gitlab³⁷ para la gestión de los proyectos software y la implementación de prácticas ágiles del CSD, de acuerdo a las pautas evaluadas en la sección 3.1, como parte de un enfoque de extremo a extremo para adoptar CD. En especial, Gitlab es capaz de ofrecer servicios de CI en una plataforma unificada, según las características establecidas en la sección 2.1.1. Además, se considera que el SIU cuenta con una instancia de Gitlab³⁸ disponible para todos los proyectos. Por todo lo expuesto, Gitlab es la plataforma sobre la cual se lleva a cabo la implementación

37 Gitlab, proyecto open source para el soporte de los equipos de desarrollo de software. Provee repositorios Git, herramienta popular entre desarrolladores usada para hacer control de versiones, Gitlab además puede integrarse como servidor de CI/CD. Esto permite automatizar todos los procesos que van desde planificación del proyecto y administración del código fuente, integración y despliegue, seguridad y monitoreo.

38 Instancia Gitlab disponible en <https://gitlab.siu.edu.ar>

del modelo de procesos definido en la sección 3.3, en particular del DP propuesto en la sección 3.2.3.

Para implementar el proceso de desarrollo de software, tal cual lo expuesto en la sección 2.3.2.2, el SIU adopta un sistema de control de versiones. La herramienta Git ya está en uso en varios proyectos que lleva a cabo el SIU. Git³⁹ es un sistema de control de versiones distribuido que fue diseñado para ser simple, rápido y soportar el desarrollo no lineal (múltiples ramas de desarrollo en paralelo). En la actualidad, existe una gran variedad de estrategias de ramificación [125], entre ellas GitFlow [126] (una implementación particular de la estrategia “branch by feature”, que permite a equipos de desarrollo trabajar en forma colaborativa y ofrece un marco de trabajo para la publicación de versiones). Según se describe en la sección 2.3.2.2, GitFlow se adapta al proceso de publicación de versiones utilizado por el SIU. Además, el DP propuesto en la sección 3.2.3, ofrece etapas y tareas que se adapten a este marco de trabajo de ramificación, tal cual se propone en la sección 3.2.3.1 y se especifica en mayor detalle en la sección 3.3.3.

Finalmente, desde hace unos años los proyectos que desarrolla el SIU hacen uso de las *tecnologías de contenedores*⁴⁰, para empaquetar aplicaciones. Al principio, para facilitar la tarea de pruebas de integración entre las aplicaciones (con un solo comando, con un ambiente altamente controlado, se despliega un conjunto de servicios y aplicaciones). En la actualidad, para proporcionar entornos de desarrollo a sus equipos (la forma de construcción y uso de contenedores junto con las herramientas involucradas, están altamente condicionadas a servir a esta tarea). El uso de la tecnología de contenedores, (Docker⁴¹ para el empaquetado y Docker Swarm⁴² para la orquestación), va en sintonía con las premisas de CD, detalladas en la sección 2.1.3.2. Esto es, Docker permite ofrecer un proceso de

39 Git, <https://git-scm.com>

40 También conocida como contenedor, es una unidad de software estandarizada. Permite empaquetar y encapsular una aplicación, con todas sus dependencias, para ser ejecutada en forma aislada de cualquier otro proceso. Este contenedor incluye lo necesario para ejecutar la aplicación: código, rutinas de ejecución, herramientas de sistema, librerías y configuración.

41 Docker permite empaquetar una aplicación en formato estándar, conocido como imagen docker. Una imagen docker es el resultado de empaquetar una aplicación para su posterior ejecución como una aplicación contenerizada. Docker, permite tomar una imagen y ponerla en ejecución, generando una instancia de dicha aplicación o contenedor.

42 Docker Swarm permite la orquestación de contenedores, incluyendo la administración del ciclo de vida de los contenedores en entornos dinámicos.

despliegue del software a producción, repetible, capaz de ser automatizado y ser confiable. En el DP propuesto en la sección 3.2.3, se ofrecen tareas específicas para el trabajo con contenedores, tal cual se plantea en la sección 3.2.3.1 y se describen en detalle en el proceso propuesto en la sección 3.3.2.

4.2 Preparación del proyecto SIU-Araí

SIU-Araí es una plataforma integradora de servicios, desarrollada por el SIU con el objetivo de ofrecer servicios transversales a todos los proyectos que componen el universo de soluciones tecnológicas que desarrolla. Dentro de esta plataforma destacan dos proyectos, **SIU-Araí: Registry** y **SIU-Araí: Usuarios** [127]. Para este TFM, se lleva a cabo la verificación del modelo de pruebas continuas propuesto sobre el proyecto **SIU-Araí: Usuarios**.

El proyecto **SIU-Araí: Usuarios** es una solución software diseñado para ofrecer una solución de Administración de Accesos e Identidad (del inglés Identity and Access Management, IAM). En esencia, es un conjunto de procesos que permiten a una organización proveer acceso seguro a datos protegidos.

4.2.1 Arquitectura

El proyecto **SIU-Araí: Usuarios** está compuesto de una serie de módulos, que ofrecen diversos servicios:

- **IdP**: módulo proveedor de identidad, ofrece Inicio de Sesión Único (del inglés Single SignOn, SSO), tales como SAML⁴³ y OIDC⁴⁴.
- **IdM**: módulo de administración de identidades y accesos digitales. Es donde se operan las configuraciones de la plataforma, se aplican políticas y se realizan el seguimiento de las mismas.
- **API**: módulo de interconexión de servicios, mediante un protocolo HTTP. Es un servicio web tipo Restful.
- **Core**: es el módulo central, alberga los mecanismos de acceso a todos los elementos internos. Es compartido entre el resto de los demás módulos.

43 Security Assertion Markup Language, <https://www.oasis-open.org/standards#samv2.0>

44 OpenID Connect, <https://openid.net/connect/>

De los cuatro módulos indicados, tres de ellos son módulos instalables de forma independiente: **IdP**, **IdM** y **API**. Es decir, cuentan con un proceso de Gestión de la Configuración y Empaquetado individual. Estos esquemas son necesarios para lograr la instalación/actualización y despliegue automatizados, tal y como se describe en la sección 2.1.3.2. A continuación se detalla en que forma genérica como se implementan estos esquemas.

4.2.2 Gestión de la configuración

La configuración de una aplicación es todo lo que puede variar entre despliegues (entornos de pruebas, preproducción, producción, desarrollo, etc.). La gestión de la configuración [128] es un componente esencial de CD, descrito en la sección 2.1.3.2 y es parte de la solución propuesta en la sección 3.2.2 apartado d). Para llevarlo a cabo, se requiere una estricta separación de la configuración y el código. La configuración puede variar entre despliegues, el código no.

El proyecto **SIU-Arai: Usuarios** permite establecer la configuración mediante variables de entorno. Las variables de entorno se modifican fácilmente entre despliegues sin realizar cambios en el código; las variables de entorno son controles granulares, cada una de ellas completamente ortogonales a las otras. El instalador de proyectos⁴⁵ utilizado por el SIU, soporta así también la configuración mediante variables de entorno.

4.2.3 Empaquetado

Para el proyecto **SIU-Arai: Usuarios**, mediante Docker, se logra empaquetar la aplicación, soportando los requerimientos de su arquitectura de componentes anteriormente detallada. El empaquetado es parte de un proceso de despliegue automatizado, es un componente esencial de CD, descrito en la sección 2.1.3.2 y es parte de la solución propuesta en la sección 3.2.2 apartado c). En el anexo sección 7.1 se ofrece en detalle las instrucciones para Docker, que permiten la construcción de las imágenes Docker que hace uso el proyecto.

45 SIU-Instalador, disponible en <https://gitlab.siu.edu.ar/siu/instalador>

El esquema de empaquetado del proyecto **SIU-Araí: Usuarios** que se hace referencia en la sección 7.1.3, se demuestra como una propuesta factible para que los proyectos del SIU provean imágenes Docker de las aplicaciones, capaces de ser puestas en marcha en ambientes de producción y apoyados en la creciente adopción de tecnologías de contenedores. **Docker Swarm** está basado en Docker, es de fácil instalación y administración, siendo ideal para pequeños y medianos ambientes de orquestación de contenedores, tales como un entorno de CI. Dentro del SIU, esta tecnología ya es conocida, por lo que será utilizada para realizar los despliegues de ambientes que sean requeridos por las diferentes fases del DP propuesto en la sección 3.2.3.

4.3 Implementación del Proceso de Pruebas Continuas

La implementación del proceso de pruebas continuas representa llevarlo a la práctica para su validación. A continuación se identifican las fases y tareas del DP implementado, finalizando con un esquema que representa como se construye cada tarea del *pipeline* en la herramienta Gitlab.

4.3.1 Selección de procesos del modelo

En la sección 3.3.1 y 3.3.2 se definieron las tareas que componen el modelo de procesos para pruebas continuas del SIU. Para llevarlo a la práctica en el proyecto **SIU-Araí: Usuarios**, se seleccionaron un conjunto de dichas tareas y fases, tal como se muestra en la Tabla 8.

Tabla 8: Selección de tareas según la fase del DP, implentadas en el proyecto SIU-Araí: Usuarios.

<i>Construcción</i>	<i>Pruebas</i>	<i>Calidad</i>	<i>Despliegue</i>	<i>Pruebas desp.</i>	<i>Publicación</i>
A1.1.1	A1.2.1	A1.3.1	A1.4.1	A1.5.2	A1.6.1
		A1.3.2		A1.5.3	
B1.1.2	B1.2.1	B1.3.1	B1.4.1		B1.6.1

Al momento de definir el ecosistema software en la sección 4.1, así como

la forma en que interactúan dichas herramientas entre sí, se estableció la importancia del manejo de ramas de desarrollo. Tal cual se propuso al momento de definir el DP en la sección 3.3.3, existen tareas que se podrán ejecutar en uno u otra rama de desarrollo, según las necesidades. La Tabla 9 muestra en que ramas de desarrollo ejecutar las tareas que previamente fueron seleccionadas para su implementación en el proyecto **SIU-Araí: Usuarios**.

Tabla 9: Tareas a ejecutar en las diferentes ramas de desarrollo del proyecto SIU-Araí: Usuarios.

	<i>feature/***</i>	<i>merge request</i>	<i>develop/master/tag</i>
A1.1.1	obligatoria	obligatoria	obligatoria
B1.1.2	obligatoria	obligatoria	obligatoria
A1.2.1	obligatoria	obligatoria	obligatoria
B1.2.1	obligatoria	obligatoria	obligatoria
A1.3.1	opcional	opcional	opcional
A1.3.2	opcional	opcional	opcional
B1.3.1	opcional	opcional	opcional
A1.4.1	no	obligatoria	obligatoria
B1.4.1	no	obligatoria	obligatoria
A1.5.2	no	manual	manual
A1.5.3	no	manual	manual
A1.6.1	no	no	obligatoria
B1.6.1	no	no	obligatoria

4.3.2 Conducto de despliegue

El conducto de despliegue o DP propuesto se implementa en la herramienta Gitlab, mediante el armado de archivos. `gitlab-ci.yml`, que permite representar el *pipeline* de un proyecto. En el anexo sección 7.2.3 se describe en detalle como se escribe uno de estos archivos. Como el proyecto **SIU-Araí: Usuarios** consiste de tres módulos, se opta por detallar en la siguiente sección las tareas para el módulo **IdP**, siendo trasladable al resto de ellos. A continuación, se presentan en forma individual cada una de las tareas seleccionadas en la sección previa, tal cual como son implementadas en Gitlab.

4.3.2.1 Etapa 1: Construcción

Tarea	A1.1.1 Construcción de las dependencias
Tipo	Obligatoria
Descripción	Ejecuta la descarga de dependencias del proyecto vía gestor de dependencias Composer.
Gitlab-ci stage	build
Gitlab-ci job	composer:idp
Gitlab-ci pipeline	<pre>composer:idp: stage: build image: siutoba/php:7.3-alpine script: - composer install --working-dir idp artifacts: paths: - ./vendor/ expire_in: 1 month</pre>

Tarea	B1.1.2 Construcción de la imagen Docker
Tipo	Obligatoria
Descripción	Realiza la construcción de la imagen docker conteniendo la aplicación y sus dependencias.
Gitlab-ci stage	build
Gitlab-ci job	docker:image-idp
Gitlab-ci pipeline	<pre> docker:image-idp: stage: build image: docker:18.09.7 extends: - docker build --pull --cache-from \$CI_IMAGE_LATEST --tag \$CI_IMAGE_APP_BUILD --file docker/Dockerfile --build-arg APP=\$CI_IMAGE_APP . - docker push \$CI_IMAGE_BUILD - docker tag \$CI_IMAGE_BUILD \$CI_IMAGE_LATEST - docker push \$CI_IMAGE_LATEST variables: CI_IMAGE_APP: idp </pre>

4.3.2.2 Etapa 2: Pruebas

Tarea	A1.1.1 Ejecución de las pruebas unitarias
Tipo	Obligatoria
Descripción	Evaluar mediante pruebas unitarias el código fuente, verificando su funcionamiento interno.
Gitlab-ci stage	test
Gitlab-ci job	unit_test:core
Gitlab-ci pipeline	<pre>unit_test:core: stage: test image: siutoba/php:7.3-alpine script: - ./phpunit -c core/phpunit.xml</pre>

4.3.2.3 Etapa 3: Calidad

Tarea	A1.3.1 Evaluación mediante análisis estático
Tipo	Opcional
Descripción	Evaluar mediante análisis estático el código fuente.
Gitlab-ci stage	quality
Gitlab-ci job	php_format:idp
Gitlab-ci pipeline	<pre>php_format:core: stage: quality image: siutoba/php:7.3-alpine script: - ./php-cs-fixer fix --dry-run diff --config idp/.php_cs allow_failure: true</pre>

Tarea	A1.3.2 Evaluación de las dependencias externas
Tipo	Opcional
Descripción	Evaluar mediante escaneo de dependencias de Composer, posibles dependencias con vulnerabilidades conocidas.
Gitlab-ci stage	quality
Gitlab-ci job	vendor_scanning:idp
Gitlab-ci pipeline	<pre> vendor_scanning:idp: stage: quality image: siutoba/php:7.3-alpine script: - ./security-checker security:check idp/composer.lock allow_failure: true </pre>

Tarea	B1.3.1 Escaneo de la imagen Docker
Tipo	Opcional
Descripción	Evaluar mediante un análisis de <i>vulnerabilidades de seguridad bien conocidas</i> , librerías que componen la imagen Docker. Usa plantilla de servicios definida por Gitlab.
Gitlab-ci stage	quality
Gitlab-ci job	container_scanning:idp
Gitlab-ci pipeline	<pre> include: - template: Container-Scanning.gitlab-ci.yml container_scanning:idp: stage: quality </pre>

4.3.2.4 Etapa 4: Despliegue

Tarea	A1.4.1 Instalación de la aplicación
Tipo	Opcional
Descripción	Ejecución para que la aplicación quede en un estado de funcionamiento correcto o adecuado, lo más cercano a un ambiente de producción. Se realiza una instalación manual.
Gitlab-ci stage	deploy
Gitlab-ci job	deploy_manual:idp
Gitlab-ci pipeline	<pre> deploy_manual:idp: stage: deploy image: siutoba/php:7.3-alpine script: - cp templates/.env .env - ./bin/instalador proyecto:instalar -n - ./bin/test-app-running.sh </pre>

Tarea	B1.4.1 Ejecución de la imagen Docker
Tipo	Opcional
Descripción	Ejecución para que la aplicación quede en un estado de funcionamiento correcto o adecuado, orquestado con <i>docker compose</i> . También verifica que haya quedado en funcionamiento.
Gitlab-ci stage	deploy
Gitlab-ci job	deploy_docker:idp
Gitlab-ci pipeline	<pre> deploy_docker:idp: stage: deploy image: gitlab.siu/siu-arai/usuarios:\$CI_COMMIT_SHA services: - postgres - ldap script: - ./bin/test-app-running.sh </pre>

4.3.2.5 Etapa 5: Pruebas despliegue

Tarea	B1.5.2 Ejecución pruebas de integración API
Tipo	Opcional
Descripción	Evaluar mediante pruebas de integración el funcionamiento de la interfaz de programación o API.
Gitlab-ci stage	post-test
Gitlab-ci job	test_api:idp
Gitlab-ci pipeline	<pre>test_api:idp: stage: post-test image: gitlab.siu/siu-arai/usuarios:\$CI_COMMIT_SHA services: - postgres - ldap script: - ./phpunit -testsuite integration</pre>

Tarea	B1.5.2 Ejecución pruebas de integración de protocolos
Tipo	Opcional
Descripción	Evaluar mediante pruebas de integración el funcionamiento de los protocolos de integración de aplicaciones cliente SAML y OIDC
Gitlab-ci stage	post-test
Gitlab-ci job	test_protocol:idp
Gitlab-ci pipeline	<pre>test_protocol:idp: stage: post-test image: gitlab.siu/siu-arai/usuarios:\$CI_COMMIT_SHA services: - postgres - ldap script: - ./bin/test-client-protocols</pre>

4.3.2.6 Etapa 6: Publicación

Tarea	B1.6.1 Publicación de la imagen Docker de la aplicación
Tipo	Obligatoria
Descripción	A la imagen generada en la fase inicial, se la etiqueta de forma adecuada a la versión de la aplicación y se la publica.
Gitlab-ci stage	publish
Gitlab-ci job	publish_docker:idp
Gitlab-ci pipeline	<pre>publish_docker:idp: stage: publish script: - docker tag \$CI_COMMIT_SHA \$CI_IMAGE_APP - docker publish \$CI_IMAGE_APP</pre>

4.4 Resultados y Discusiones

Gracias a las herramientas de analítica para CI que provee Gitlab⁴⁶, fué posible adquirir información de la ejecución de los componentes CI, que son generadas automáticamente de los datos en Git. Dos de los hallazgos más significantes incluye el *tiempo de ejecución de pipelines* y el *porcentaje de éxito de pipelines*. Un *pipeline* posee muchas tareas, de cuya ejecución se obtiene el resultado para evaluar el suceso o falla del mismo. Fowler indica que una de las prácticas claves en CI, es que el equipo pueda realizar *commit* múltiples veces al día [26], y esto significa que múltiples *pipelines* serán procesados en el mismo día. Calcular el tiempo requerido por cada *pipeline*, junto con el éxito de su ejecución, permite entender la efectividad de esta práctica.

4.4.1 Pruebas que fueron automatizadas

Cada una de las tareas ejecutadas en el *pipeline* permiten reaccionar al resultado de su ejecución. Si bien algunas tareas no son un tipo de prueba como

46 The DevOps Lifecycle with GitLab. <https://about.gitlab.com/stages-devops-lifecycle/>

se las identificó en la sección 2.1.4, sirven para verificar que la aplicación pueda funcionar correctamente. Se realizaron dos tareas de este tipo:

- construir las dependencias de forma correcta
- empaquetar la aplicación en imagen docker

Posteriormente, se ejecutaron las pruebas del tipo unitaria, de integración, y de funcionamiento tanto GUI como CLI:

- pruebas unitarias
- pruebas de despliegue mediante instalación manual
- pruebas de despliegue mediante imagen docker
- pruebas de integración de API
- pruebas de integración para protocolos SAML y OIDC

El resultado de cada una de estas pruebas está disponible en Gitlab mediante el acceso a los resultados de la ejecución de los *pipelines*. En la siguiente sección se detalla cómo se ofrece esta retroalimentación.

4.4.2 Retroalimentación

En la sección 2.1.1.1, cuando se describe el proceso de CI, se identifica al sistema de *feedback* como un componente prioritario al realizar las integraciones: permite notificar lo más rápido y preciso posible el resultado de la ejecución del DP. Gitlab ofrece en su plataforma, de manera integrada, diversos mecanismos que permiten conocer los resultados en la ejecución de un *pipeline*. Uno de ellos es el “Tablero de Pipelines”, que muestra los resultados obtenidos en la ejecución de los últimos *pipelines*. La Figura 33 muestra dicho tablero, con un resumen de los resultados obtenidos: éxito, *commit* y desarrollador que lo disparó y duración.

siu-arai > arai-usuarios > Pipelines

Todos 807 Pendiente 0 En ejecución 0 Finalizado 800 Ramas Etiquetas

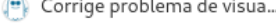
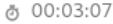
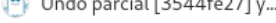
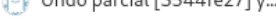
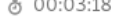
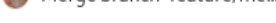
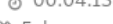
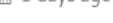
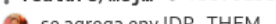
Estado	Pipeline	Disparador	Commit	Etapas
	#6356		  	 00:03:07 
	#6354 Desvinculado		  	 00:00:53 
	#6353		  	 00:03:18 
	#6352		  	 00:04:13 
	#6351		  	 00:07:53 
	#6350		  	 00:07:07 

Figura 33 Tablero de Pipelines, con resultados de los últimos ejecutados

Gitlab usa tres colores para identificar el estado de un *pipeline* y el de cada una de sus etapas y tareas: el **verde** significa que la tarea se completó con éxito; en **naranja**, la tarea no tuvo éxito pero se permite fallar; en **rojo**, la tarea no tuvo éxito y no se permite el fallo. Adicionalmente, el resultado final del *pipeline* se muestra en **verde** si todas las tareas se ejecutaron con éxito, en **naranja** si alguna tarea no crítica tuvo un error, **rojo** si alguna tarea considerada crítica ha fallado.

Gitlab ofrece, desde el “Tablero de Pipelines”, la posibilidad de acceder a los detalles del resultado arrojado por cada una de las tareas que componen el *pipeline*. La Figura 34 muestra en detalles el resultado de un *pipeline*, incluyendo:

- indicadores en colores del estado
- información descriptiva del *commit* en Git que disparó el *pipeline*.
- duración de la ejecución total del *pipeline*, de manera global como por trabajos.

- representación gráfica de las etapas y tareas que componen el *pipeline* que fue ejecutado.
- posibilidad de relanzar la ejecución de las tareas
- visualizar la salida que produce la tarea en la consola de ejecución

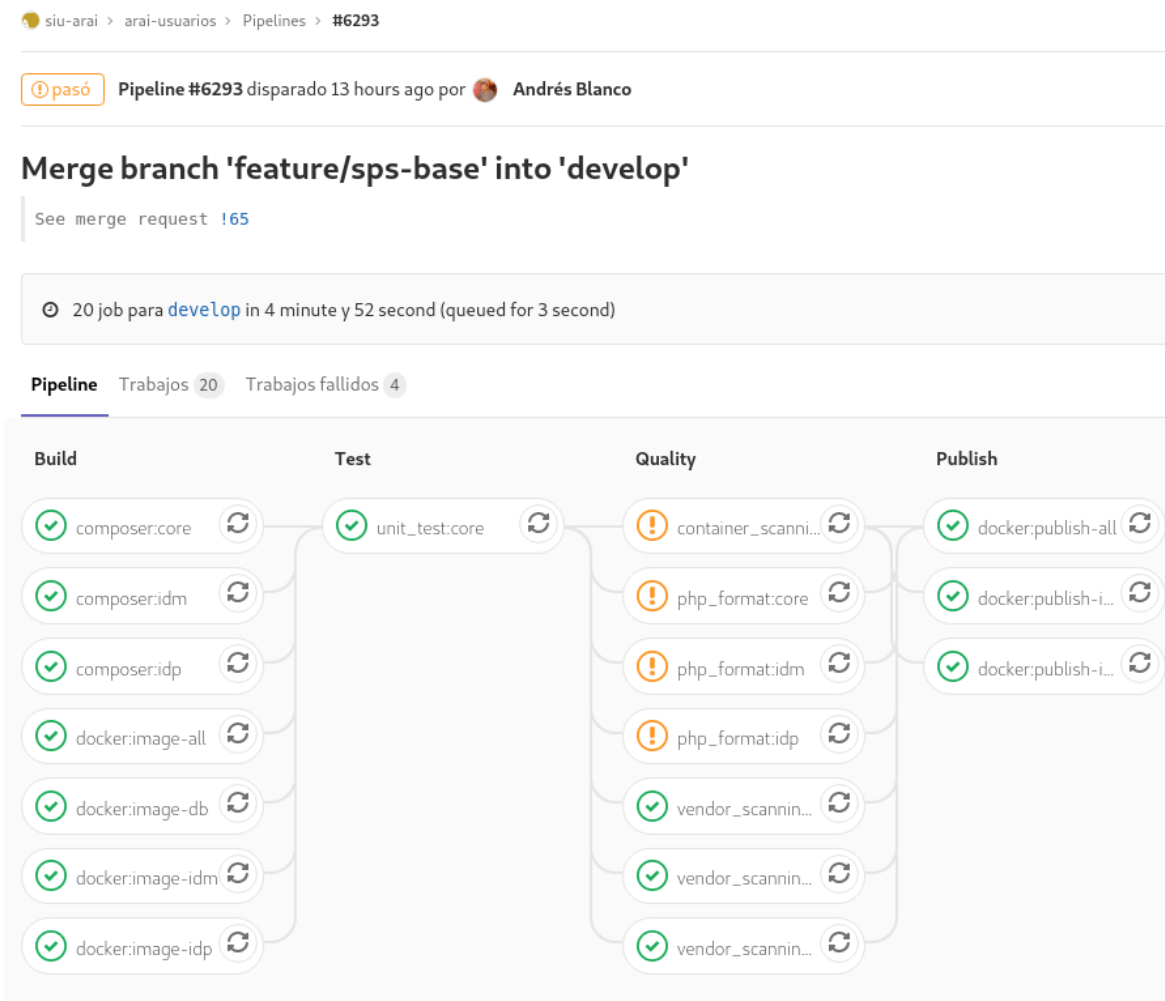


Figura 34 Visualización de los resultados de las tareas de un pipeline

De acuerdo a lo propuesto en la sección 3.3.3 como parte de la solución, el DP implementado debe adaptarse al proceso de desarrollo del proyecto. En la sección 4.1 se describió la forma en que se organiza el flujo de desarrollo en el proyecto SIU-Araí, mediante ramas de desarrollo en Git y el uso de la herramienta Gitflow. En resumen, mientras se está realizando *commits* sobre una rama de desarrollo en particular, y antes de finalizarla, el *pipeline* que se ejecuta es mucho más reducido que uno normal, tal cual se define en la sección 4.3.1. En la Figura 35

se observa cómo es dicho *pipeline*, ejecutado para una rama de desarrollo en Git denominada **feature/uid-to-uniqueIdentifier**.

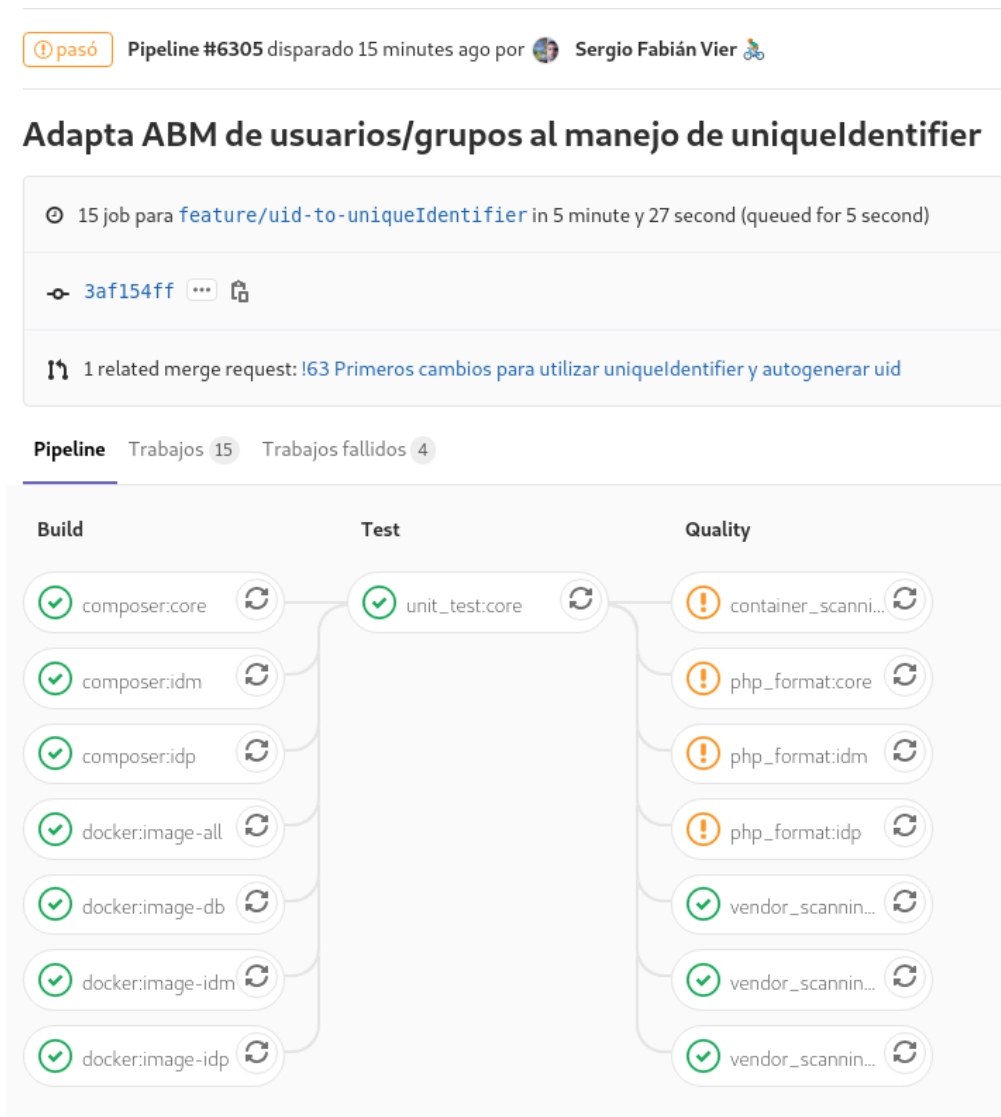


Figura 35 Visualización de los resultados de un pipeline de tareas reducido

Como parte de la integración de Git, Gitlab ofrece la funcionalidad de “merge request” (MR) o petición de fusión. Esta característica entra en juego al plantearse un esquema de desarrollo utilizando Gitflow, tal cual se describe en la sección 4.1. Un MR se crea en el momento previo al intentar fusionar una rama de característica hacia la rama principal, de modo tal que pueda darse lugar a un proceso previo de revisión de código por pares, visualización del estado del *pipeline* relacionado con la característica, etc. En la Figura 36 se observa un MR realizado en el proyecto **SIU-Araí: Usuarios**.

siu-arai > arai-usuarios > Merge Requests > !68

Abrir Opened hace 6 días by Sergio Fabián Vier

Edit Close merge request

Modularización de API

Resumen 15 Cambios 6 Pipelines 12 Cambios 100

Se modulariza el componente `api`, con identidad propia.

- creandose su propia imagen Docker
- api v1 y v2
- suite de pruebas de integración

Implementa #176 que surge de #161

Solicitud de merge `feature/api-v2` en `develop`
la rama de origen está 7 commits detrás en relación a la rama de destino

Abrir en Web IDE Check-out

Pipeline #6418 pasó con advertencias para 35d53ef9 en `feature/api-v2`

Aprobar adicionalmente merge request aprobado. Aprobado por

Ver aprobadores elegibles

Merge ☒ Eliminar rama origen ☐ Squash commits

6 commits y 1 merge commit se añadirán a `develop`. [Modificar el merge commit](#)

Figura 36 Visualización de un MR en Gitlab, con información del pipeline

4.4.3 Estadísticas

Otro de los componentes de Gitlab vinculado con CI es el de datos estadísticos e históricos de su ejecución⁴⁷ en la plataforma, realizando actividades de analítica sobre los datos de Git en forma complementaria. Permite visualizar información relativa a pipelines, MR, *commits*, *issues* y productividad en general. La forma en que Gitlab recoge esta información varía de acuerdo a lo que se esté

47 Analítica en Gitlab, <https://docs.gitlab.com/ce/user/analytics/>

visualizando, pudiendo ir a partir de la ejecución del primer *pipeline* del proyecto, los *pipelines* ejecutados durante los últimos doce meses o en el último mes.

Para el proyecto **SIU-Araí: Usuarios**, el *pipeline* inicial fué definido el 27 de abril 2018. Su armado e implementación fue realizada en forma gradual, encontrando mejoras y cambios en el transcurso de la elaboración de este TFM. Las estadísticas aquí analizadas, consideran el último *pipeline* al ejecutado el 25 de abril 2020. En la Figura 37 se muestra las ejecuciones de *pipelines* global para el proyecto, junto con el ratio de éxito (pipelines exitosos / pipelines total).

- Total: **860 pipelines**
- Exitosos: **558 pipelines**
- Fallidos: **224 pipelines**
- Ratio de éxito **71%**

Figura 37 Cantidad de pipelines ejecutados

Los *pipelines* corresponden a todos los ejecutados por el proyecto **SIU-Araí: Usuarios**, iniciados por *commits* en las ramas del repositorio *master* y *develop*, así como las *feature/** y los iniciados por los MR respectivamente. El tiempo de ejecución promedio en todos ellos, es un factor de relevancia. La Figura 38 muestra una gráfica resumiendo el tiempo de ejecución de los últimos 30 *pipelines* ejecutados

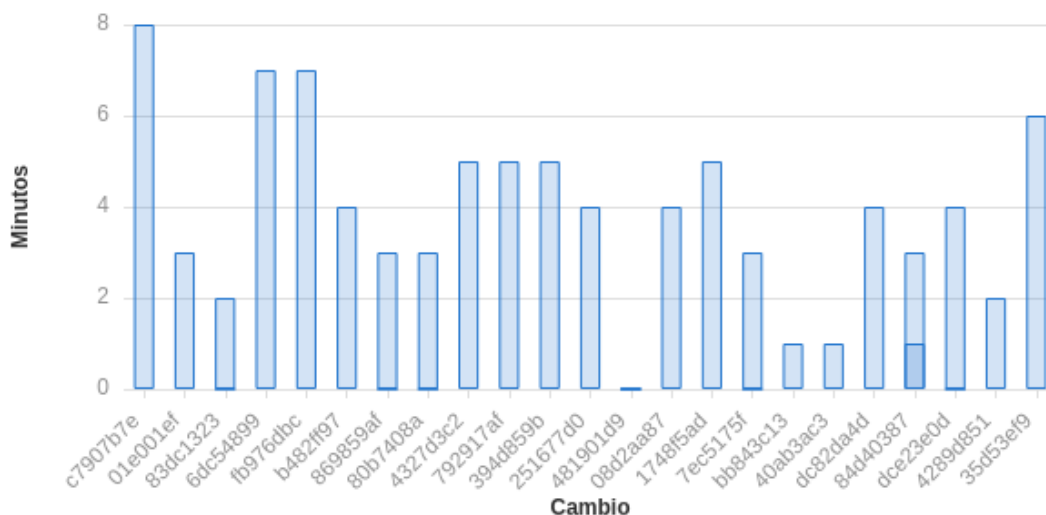


Figura 38 Duración de los últimos 30 pipelines (a 25 de abril de 2020)

El rango de tiempos de duración va de 1 minuto a los 8 minutos, con una

media de 4 minutos. El tiempo varía claramente dependiendo si es un *pipeline* reducido que se ejecuta sobre una rama *feature/** de característica (mientras se desarrolla) o si es sobre la rama *develop* o *master*, las que integran constantemente código en funcionamiento.

En la Figura 39 y Figura 40 se observa el ritmo de ejecución de *pipelines* a lo largo del último mes y del último año respectivamente. Esta gráfica expone el ratio de éxito en la ejecución de los *pipelines* sobre una frecuencia diaria y mensual respectivamente. Sobre color azul destaca los *pipelines* que hay fallado en su ejecución o fueron manualmente cancelados, incluso en menor medida los que fueron salteados mediante opciones de ejecución.

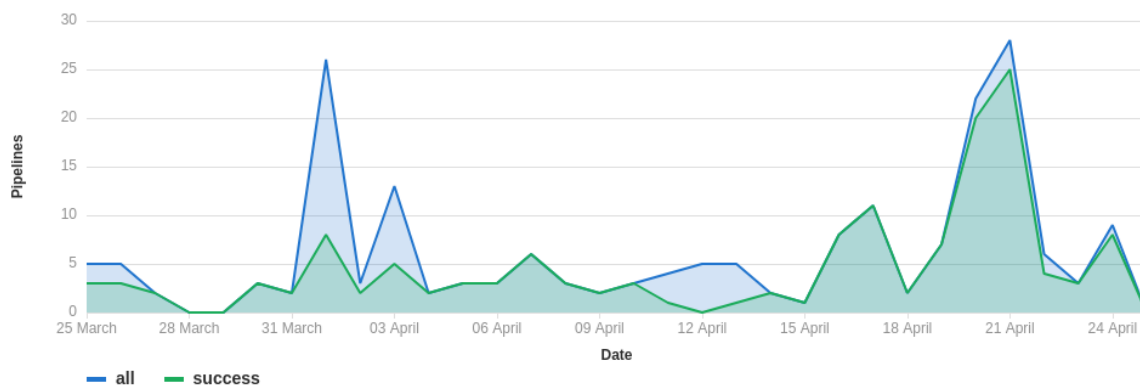


Figura 39 Pipelines ejecutados en el periodo del mes (al 25 de abril de 2020)

En la Figura 39 Pipelines ejecutados en el periodo del mes (al 25 de abril de 2020) se identifica una caída de ejecuciones exitosas entre los días 10 de abril y 14 de abril) sobre una rama *feature/**. Dicho error en la ejecución del *pipeline* se extendió por varios días, siempre con código sobre una rama para desarrollo de una característica. El *pipeline* logró detectar errores reales introducidos al producto, en este caso sobre el mecanismo de despliegue del mismo, antes de que estos errores lleguen a las ramas estables del mismo.

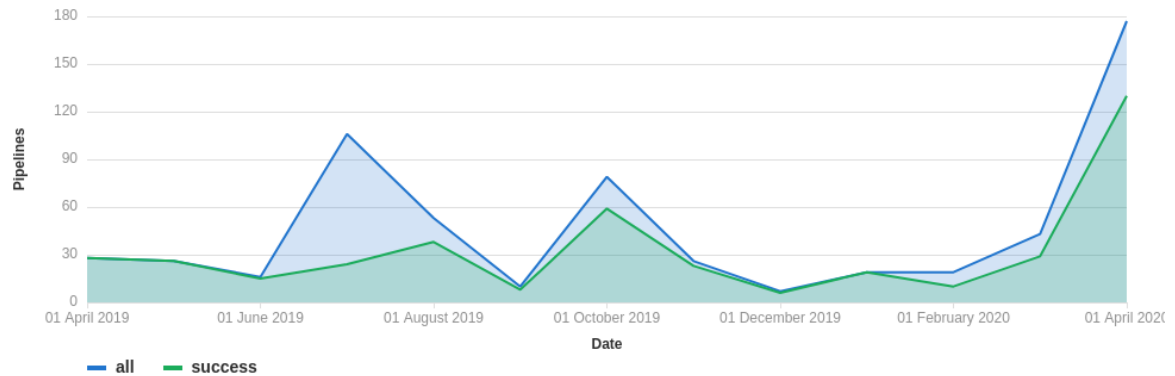


Figura 40 Pipelines ejecutados en el periodo del año (al 25 de abril de 2020)

Un motivo para fallo o cancelación de los *pipelines*, normalmente fue por la misma acción de mejora/ajustes que se fueron introduciendo a medida que se fue elaborando la estrategia de ejecución del DP. Esto se ve claramente reflejado en la Figura 40 con los 3 picos azules, de forma en punta y con duración de un día, se corresponden a actividades de modificación y prueba del *pipeline* llevados a cabo el 1 de julio de 2019, el 1 de octubre de 2019 y el 1 de abril de 2020.

4.4.4 Discusión de los resultados

Los resultados obtenidos, permiten analizar la utilidad del DP propuesto e implementado para el proyecto **SIU-Araí: Usuarios** mediante Gitlab. El número de éxitos/fallas en la ejecución de *pipelines* (visto en la Figura 37) está en el rango de resultados esperados, correspondiendo a las distintas ramas de desarrollo del producto. A pesar de los picos de error que se dieron en las pruebas de configuraciones, se logró detectar errores reales introducidos al producto (ver por ejemplo, en la Figura 39 entre los días 10 de abril y 14 de abril) sobre una rama *feature/**. Gracias a ello, dicho error no llegó a propagarse sobre las ramas estables *develop* y sobre todo *master*. En resumen, el DP propuesto y llevado a la práctica en el proyecto **SIU-Araí: Usuarios** logró introducir validaciones automáticas, tal es el caso del mecanismo de despliegue que utiliza dicha aplicación, que sirvieron para la detección efectiva y temprana de errores en dicho mecanismo de despliegue.

El tiempo consumido para completar la ejecución de un *pipeline* es una métrica importante [27] para entender con qué velocidad un desarrollador puede obtener retroalimentación de la característica que se encuentra desarrollando. *Fowler* [26] claramente expone que el objetivo de CI es proporcionar retroalimentación rápida. El DP propuesto en la sección 3.3 y llevado a la práctica como se describe en la sección 4.3 logra resultados en los tiempos de ejecución de los *pipelines* con una media de 4 minutos (ver la Figura 38), dentro del rango de los 10 minutos propuestos en las guías de XP [86]. Un objetivo del DP propuesto ha sido ofrecer distintas configuraciones de las tareas a ejecutar, logrando que la retroalimentación sea tan rápida como 2 minutos y propiciando que los desarrolladores realicen *commits* con mayor frecuencia. Claramente, este TFM no profundiza en cuestiones de competencia y disponibilidad de los recursos requeridos por múltiples *pipelines* que puedan ejecutarse en simultáneo, cuando se intenta expandir la base de proyectos que adoptan estrategias de CI/CD en el SIU.

En un entorno de CI, la cantidad de pruebas ejecutadas es otra métrica importante [27], junto con su capacidad de detectar tantos errores como sea posible. Las pruebas unitarias en particular, con sus resultados, son expuestas como resultados sobre la interfaz de MR, vista en la Figura 36. Es decir, Gitlab espera que al momento de realizar la revisión de código, los resultados de las pruebas unitarias sirvan de retroalimentación del estado de la característica que se está evaluando. En el DP implementado para el proyecto **SIU-Araí: Usuarios**, se introdujo un set de pruebas unitarias para el módulo compartido, y dos pruebas de despliegue para los módulos instalables, todas automatizadas en su ejecución. En ambos casos, se avanzó con la automatización de las pruebas que fueron viables considerando las características del actual ambiente de CI/CD disponible en el SIU. No se ha podido automatizar aún otras pruebas que dispone el proyecto: una suite de casos de pruebas sobre la API Rest, una suite de pruebas sobre protocolos de autenticación SAML y OIDC ofrecidos por el proyecto. Estas pruebas requieren el despliegue a demanda y en forma automatizada de ambientes de ejecución con una versión del proyecto, lo cual se plantea en la sección 4.1 mediante la orquestación de contenedores. Debido a requerimientos técnicos inherentes a dicha tecnología, su implementación dentro del SIU no ha

alcanzado un punto de madurez para incluirlo dentro del proceso de despliegue automatizado. Es por ello que en este TFM no se avanzó sobre el aprovisionamiento automático de entornos de pruebas, una de las funcionalidades ofrecidas por Gitlab e integrada en el uso de los MR.

Gitlab como herramienta, logra facilitar la implantación y adopción de prácticas de desarrollo continuo en general, y prácticas de pruebas continuas en particular. A pesar de ello, Gitlab aún presenta algunos inconvenientes importantes para un ambiente CI/CD. Uno de ellos es la visualización de la relación y/o dependencias que poseen entre sí las tareas de un *pipeline*. El DP descrito en 4.3.1 visualmente no logra aún⁴⁸ ser representado por Gitlab. Otro inconveniente que presenta Gitlab actualmente, es la falta de un mecanismo para establecer políticas de uso sobre repositorio de los proyectos, en la línea de evitar que los desarrolladores utilicen opciones para evitar la ejecución de los *pipelines* en determinadas ramas del repositorio.

Finalmente, como un análisis del contraste entre el antes y después en la adopción de un proceso de pruebas continuas para el proyecto **SIU-Araí: Usuarios**, se observa un impacto sustancial en diversas áreas que involucran tanto al producto como al equipo de trabajo. Desde el punto de vista del producto, lo que mayor impacto genera es la identificación casi inmediata de errores y fallas que se introducen en el desarrollo. Antes, era necesaria las pruebas manuales que involucraban la instalación y puesta a punto manual del producto, para luego ejecutar diversos tipos de verificaciones, mayormente manuales, con el fin de identificarlas. Como referencia, la Figura 41 muestra los tiempos típicos requeridos para el despliegue. Debido a la complejidad de los componentes que conforman el producto, sólo el tiempo de despliegue manual involucra tareas de 30 minutos de duración, contrastando con un despliegue automatizado mediante tecnología de contenedores, que se realiza en 2 minutos.

48 Reporte en Gitlab para mejorar la visualización del gráfico de dependencias entre tareas y etapas. <https://gitlab.com/gitlab-org/gitlab/-/issues/20096>

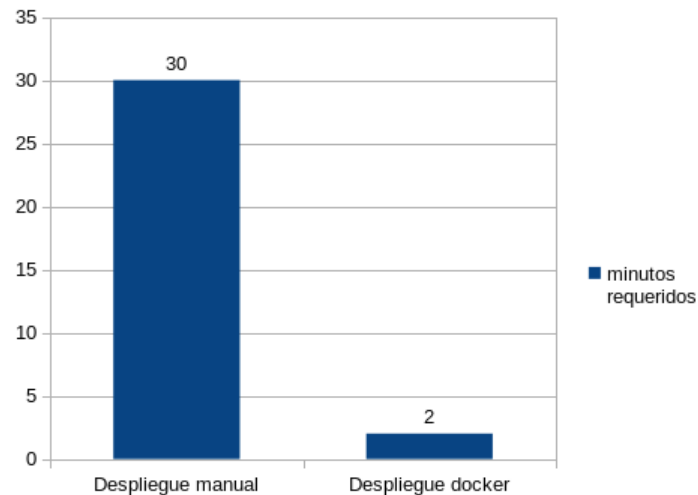


Figura 41 Contraste del tiempo requerido para el despliegue

Una reducción tan drástica en los tiempos de despliegue, posibilitó que el equipo aumente la velocidad en la que se introducen cambios al producto y su posterior verificación. El tiempo de despliegue tiene consecuencia directa en la distribución de los errores reportados durante el tiempo de desarrollo de una versión del producto, como se muestra en la Figura 42. La versión 2.3.0 fué desarrollada con el esquema existente, mientras que la versión 3.0.0 aplica el DP propuesto en este TFM.

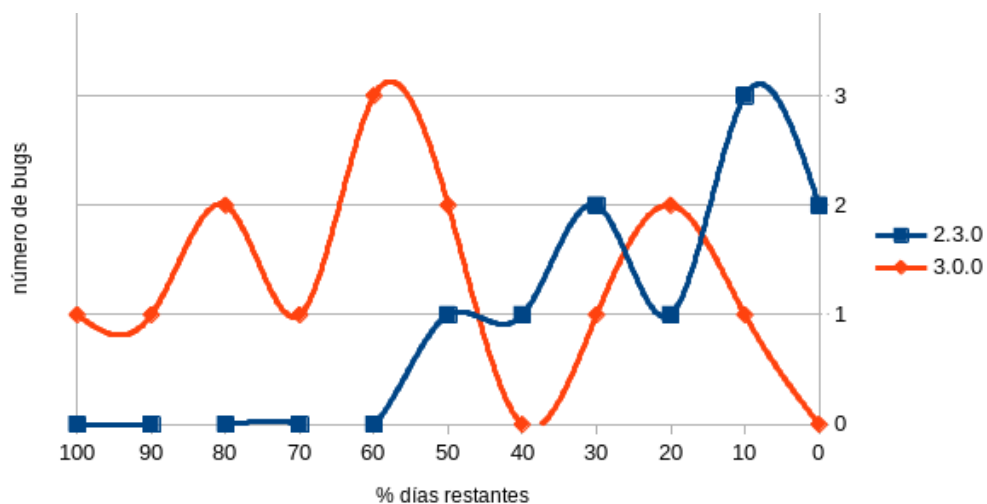


Figura 42 Distribución de errores detectados a lo largo del desarrollo

Si bien la versión 2.3.0 tuvo menos errores detectados respecto a la 3.0.0 debido al número de características involucradas en cada una, claramente se

observa que en el desarrollo asistido por el DP propuesto, los errores son detectados en la medida que se introducen nuevas características y no al final del cierre de la versión. Esto es el resultado de la aplicación de pruebas durante todo el ciclo de desarrollo y no solo al finalizar, junto con la inmediata visibilidad que tienen dichos incidentes.

Desde el punto de vista del producto, guiado por la necesidad de implementar un proceso de pruebas continuas, fue la necesaria transformación de una arquitectura monolítica a una modular. Este cambio fue impulsado para mejorar la automatización del despliegue del producto, pero también permitió la introducción de pruebas, difíciles de realizar hasta ese entonces. Esto claramente tuvo un impacto sobre el proyecto, que busca ahora introducir mayores tipos de pruebas automatizadas para mejorar y garantizar la calidad final del producto.

Desde el punto de vista de las personas que integran el equipo de trabajo, la adopción de un proceso de pruebas continuas brindó retroalimentación casi en tiempo real sobre actividades que no se llevaban a cabo con tanta frecuencia, tal es el caso del despliegue del producto. Sumado a la automatización de otras pruebas, significó un impacto positivo sobre la confianza del equipo de desarrollo para la introducción de cambios, pero sobre todo permitió que se comprometiera a mantener al producto estable, vínculo directo con el resultado de la ejecución del DP. Por el contrario, la complejidad del trabajo a realizar se vio incrementada, consecuencia de las nuevas tecnologías adoptadas, como de los cambios en el proceso de desarrollo introducidos. Cambios durante el desarrollo y que antes no se coordinaban, ahora requieren comunicación adecuada a los participantes adecuados, de lo contrario puede ser un punto de fricción tratar de mantener el estado correcto de la ejecución del DP.

5

CONCLUSIONES

5 CONCLUSIONES

En este capítulo se resumen las principales contribuciones de este TFM, analizando el cumplimiento de los objetivos establecidos en la sección 1.2. Además se analizan las posibles líneas futuras de investigación y trabajo.

5.1 Análisis del cumplimiento de los Objetivos

Al inicio de este TFM se estableció una hipótesis de trabajo: *es factible elaborar una metodología aplicable al desarrollo de software en el SIU, que permita la construcción de un proceso de pruebas continuas, así como la propuesta de un marco tecnológico para implementar y dar soporte a dicho proceso de pruebas continuas*. A continuación procede un análisis del cumplimiento de cada objetivo específico, en el marco de la hipótesis general.

El objetivo específico 1 fue definido como: *Revisión de la literatura referente al estado de situación de las técnicas de pruebas continuas en el marco de la tecnología con la que se construye la plataforma SIU-Araí*.

El desarrollo de software en la actualidad abraza un paradigma conocido como *Desarrollo Continuo de Software*. La comprensión y entendimiento de las prácticas continuas que lo componen fué esencial para abordar esta investigación (sección 2.1). Además, en el marco de este TFM se ha realizado un *Estudio Terciario de la Literatura*, que resume las revisiones sistemáticas existentes en torno a temas relacionados con la CI (sección 2.2). Este estudio permitió lograr un mejor entendimiento de las soluciones de CI y los desafíos que enfrenta una organización al momento de su adopción (sección 2.2.3). Para todos los estudios que fueron analizados, uno de los tópicos más importantes representa la *Automatización de las Pruebas*. No es coincidencia que también se describe como una de las mayores dificultades que se reportan al momento de aplicar prácticas continuas. Todo esto permitió lograr un mayor entendimiento en los desafíos para la construcción y adopción de un proceso de pruebas continuas en el SIU (secciones 2.1.5 y 2.2.3).

El objetivo específico 2 fue definido como: *Análisis y evaluación del proceso actual de desarrollo de software dentro del Sistema de Información Universitaria.*

Con el fin de lograr que la metodología que propone este TFM sea adaptable al SIU, fue necesario el análisis de los procesos de desarrollo que lleva adelante el mismo. Esto permitió comprender la forma en que el SIU aplica su metodología de desarrollo, centrando el análisis en torno a los proyectos de software y los equipos de trabajo (secciones 2.3.1 y 2.3.2). Parte del análisis estableció, entre otras cuestiones, tipos y formas de ejecución de las pruebas de software y la manera en que se llevan a cabo (sección 2.3.3). La comprensión de las pruebas de software y la integración en cada proyecto fue esencial al momento de establecer las principales características del proceso de pruebas continuas propuesto para el SIU (sección 3.2.2).

El objetivo específico 3 fue definido como: *Diseño de los procedimientos para construir una metodología personalizada de pruebas continuas, aplicables al ámbito específico del Sistema de Información Universitaria.*

Una vez determinado el marco teórico, sustento de este TFM, se estableció de manera precisa la metodología para la construcción de un proceso de pruebas continuas, como parte integral de un conjunto de prácticas continuas (sección 3.1). Previamente, del análisis del proceso de desarrollo de software adoptada por el SIU, se estableció las fases que conforman el patrón de DP propuesto (sección 3.2.3). Con el flujo de CI establecido, se diseñaron los procesos y las actividades que componen el flujo. De esta manera, se logró definir el *Modelo de procesos para pruebas continuas en el SIU* (sección 3.3). Complementariamente, el análisis previo de la organización del SIU estableció que cada proyecto presenta particularidades, por lo que el modelo propuesto ha de ser adaptable a estas necesidades (sección 3.3.3).

El objetivo específico 4 fue descrito como: *Definición y configuración de un ecosistema software que permita dar soporte a dicho conjunto de procedimientos,*

en un marco de desarrollo de software moderno y adaptable.

Para implementar la plataforma de pruebas continuas, se utilizaron herramientas que ya estaban en proceso de adopción dentro del SIU. La solución se basa en el uso de la plataforma Gitlab y su componente GitlabCI. Dicha plataforma permite interactuar con otras tecnologías, algunas en diferentes grados de adopción dentro del SIU dependiendo del proyecto, tales como Git, Gitflow, Docker, entre otros. Finalmente, la definición de procesos de construcción y ejecución de pruebas automatizables, contemplados en el modelo de procesos de pruebas continuas propuesto (sección 4.1), se realizaron de acuerdo a las funcionalidades que ofrece Gitlab como plataforma unificada.

El objetivo específico 5 fue definido como: *Validación de los procedimientos propuestos con la configuración del ecosistema software, utilizando como caso de estudio el desarrollo de la plataforma SIU-Araí.*

La validación del modelo de procesos para pruebas continuas en el SIU fue realizado sobre el proyecto **SIU-Araí: Usuarios**. Dicho proyecto ofrece las características para ser adaptado al esquema de pruebas y despliegue automatizados (sección 4.2). Para lograr la implementación del modelo, se seleccionó una serie de procesos, etapas y actividades (sección 4.3.1) que fueron utilizados en el DP para **SIU-Araí: Usuarios** (sección 4.3.2). Su implementación permitió obtener retroalimentación con el desarrollo real del proyecto (sección 4.4.2), reflejando el impacto que produce la automatización en la ejecución y validación de pruebas de software. El proyecto **SIU-Araí: Usuarios** mejoró la forma de realizar el despliegue (antes una tarea manual de 30 minutos y ahora una tarea de 2 minutos de fácil automatización), introdujo nuevas pruebas de software específicas sobre los módulos y expuso errores de desarrollo de manera temprana en el ciclo desarrollo del proyecto.

5.2 Conclusiones generales

En base al trabajo realizado se ha observado que la adopción de un

proceso de pruebas continuas requiere embarcarse en una serie de cambios, tanto del producto como del equipo de trabajo. Es una transición desde un modelo tradicional a uno continuo. Por otro lado, el impacto que se obtuvo de la aplicación de un conducto de despliegue, medio para la automatización de las pruebas, ha sido positivo y reflejó resultados inmediatos. Es posible afirmar que quedan establecidas la bases para contar con una plataforma de pruebas continuas en el desarrollo de software del SIU.

5.3 Trabajo Futuro

En el camino recorrido en este TFM se han desvelado muchas características que hacen al desarrollo continuo de software y a las pruebas continuas en particular. No todas ellas pueden contemplarse, incluso pueden requerir el logro de hitos previos para poder considerarlas. A continuación se describen las líneas futuras que son factibles para su investigación y/o implementación.

5.3.1 Expandir el ámbito de aplicación de las pruebas continuas

En el DP propuesto, se han contemplado las fases que hacen al desarrollo del producto software, pero no así a las que hacen a su operación. Existen ejemplos [129] de como las pruebas continuas facilitan la entrega continua. Un camino para lograrlo es que la fase operación del producto pueda reutilizar las pruebas existentes, tanto en producción como en actividades de monitoreo.

5.3.2 Mejora de la evaluación de las pruebas en la fase de desarrollo

La plataforma Gitlab ofrece un mecanismo para revisión de código, denominado *Merge Request*, actividad ligada al momento en que se finaliza el desarrollo de una característica. La plataforma abre camino a que en dicho punto y de forma automática, se produzca de forma automatizada despliegues y pruebas sobre dicha característica en desarrollo.

5.3.3 Optimizar la ejecución del Pipeline

No se ha profundizado en este TFM las cuestiones vinculadas al desempeño del DP propuesto bajo una gran carga de trabajo (conurrencia de DP en ejecución, concurrencia entre proyectos con sus DP respectivos), ni la priorización de tareas o la segmentación en la ejecución de casos de pruebas. La optimización y la escalabilidad de las tareas en el DP propuesto, dependerán de la puesta a punto de la plataforma Gitlab.

6

REFERENCIAS

6 REFERENCIAS

- [1] L. Gurmendi, «El Sistema de Información Universitaria», 1998. [En línea]. Disponible en: <http://repositorio.educacion.gov.ar:8080/dspace/handle/123456789/93088> (accedido jul. 24, 2017)
- [2] «SIU – Sistema de Información Universitaria», 2020. [En línea]. Disponible en: <https://www.siu.edu.ar/> (accedido ene. 06, 2020).
- [3] G. M. Falivene, M. de Luján Gurmendi, y G. M. Silva, «El e-learning como mecanismo articulador de procesos de gestión del conocimiento y formación continua en las organizaciones públicas: El caso del Sistema de Información Universitaria (SIU)».
- [4] «Sistema de Información Universitario - Nuestra gestión en números», *Nuestra gestión en números*. <http://www.siu.edu.ar/nuestra-gestion-en-numeros/> (accedido ene. 06, 2020).
- [5] «Documentación - SIU-Arai». <http://documentacion.siu.edu.ar/wiki/SIU-Arai> (accedido ene. 06, 2020).
- [6] J. Järvinen, T. Huomo, T. Mikkonen, y P. Tyrväinen, «From agile software development to mercury business», en *International Conference of Software Business*, 2014, pp. 58–71. [En línea]. Disponible en: http://link.springer.com/10.1007/978-3-319-08738-2_5. (accedido: jul. 25, 2017)
- [7] 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.
- [8] 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.
- [9] 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.
- [10] 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.
- [11] F. Khomh, T. Dhaliwal, Y. Zou, y B. Adams, «Do Faster Releases Improve Software Quality?: An Empirical Case Study of Mozilla Firefox», en *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories*, Piscataway, NJ, USA, 2012, pp. 179–188. [En línea]. Disponible en: <http://dl.acm.org/citation.cfm?id=2664446.2664475>. (accedido: jul. 24, 2017)
- [12] 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.
- [13] 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.
- [14] 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.
 - [15] J. Humble y D. Farley, *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Pearson Education, 2010.
 - [16] K. Beck et al., «Manifesto for agile software development», 2001. [En línea]. Disponible en: <http://nlp.chonbuk.ac.kr/SE/ch05.pdf>. (accedido: jul. 24, 2017)
 - [17] I. C. Society, P. Bourque, y R. E. Fairley, *Guide to the Software Engineering Body of Knowledge (SWEBOK(R)): Version 3.0*, 3rd ed. Los Alamitos, CA, USA: IEEE Computer Society Press, 2014.
 - [18] E. Marcos y A. Marcos, «An Aristotelian Approach to the Methodological Research: a Method for Data Model Construction», *Inf. Syst. Gener. McGraw Hill*, pp. 532–543, 1999.
 - [19] E. Marcos, G. Kybele, y J. Carlos, «Investigación en Ingeniería del Software vs. Desarrollo Software», ene. 2020.
 - [20] M. Bunge, *La investigación científica: su estrategia y su filosofía*. Siglo XXI, 2002.
 - [21] T. O'Reilly, *WTF?: What's the Future and Why It's Up to Us*. HarperCollins, 2017.
 - [22] 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.
 - [23] G. Booch, *Object solutions: managing the object-oriented project*. Addison Wesley Longman Publishing Co., Inc., 1995.
 - [24] S. McConnell, *Software Project Survival Guide: How to Be Sure Your First Important Project Isn't Your Last*. Microsoft Press, Redmond, WA, 1998.
 - [25] P. M. Duvall, S. Matyas, y A. Glover, *Continuous integration: improving software quality and reducing risk*. Pearson Education, 2007.
 - [26] M. Fowler, «Continuous Integration», *martinfowler.com*. <https://martinfowler.com/articles/continuousIntegration.html> (accedido ene. 15, 2020).
 - [27] N. Vafaie y M. Arvidsson, «Metrics to measure the impact of continuous integration-An empirical study», 2015.
 - [28] J. Biolchini, P. G. Mian, A. C. C. Natali, y G. H. Travassos, «Systematic review in software engineering», *Syst. Eng. Comput. Sci. Dep. COPPEUFRJ Tech. Rep. ES*, vol. 679, n.º 05, p. 45, 2005.
 - [29] P. Clements y L. Northrop, *Software Product Lines: Practices and Patterns: Practices and Patterns*. Addison-Wesley, 2015.
 - [30] 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, pp. 56–65, 2010.
 - [31] T. Coupaye y J. Estublier, «Foundations of enterprise software deployment», en *Proceedings of the Fourth European Conference on Software Maintenance and Reengineering*, 2000, pp. 65–73.
 - [32] H. H. Olsson, H. Alahyari, y J. Bosch, «Climbing the "Stairway to Heaven"—A Multiple-Case Study Exploring Barriers in the Transition from Agile Development towards Continuous Deployment of Software», en *2012 38th euromicro conference on software engineering and advanced applications*,

- 2012, pp. 392–399.
- [33] T. Dyb\aa y T. Dings\oyr, «Empirical studies of agile software development: A systematic review», *Inf. Softw. Technol.*, vol. 50, n.º 9-10, pp. 833–859, 2008.
- [34] R. C. Martin, *Agile software development: principles, patterns, and practices*. Prentice Hall, 2002.
- [35] M. Poppendieck y T. Poppendieck, *Lean Software Development: An Agile Toolkit: An Agile Toolkit*. Addison-Wesley, 2003.
- [36] L. Bocock y A. Martin, «There’s something about lean: A case study», en *2011 Agile Conference*, 2011, pp. 10–19.
- [37] P. Agarwal, «Continuous SCRUM: agile management of SAAS products», en *Proceedings of the 4th India Software Engineering Conference*, 2011, pp. 51–60.
- [38] 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.
- [39] F. J. Lacoste, «Killing the gatekeeper: Introducing a continuous integration system», en *2009 agile conference*, 2009, pp. 387–392.
- [40] T. Van Der Storm, «Backtracking incremental continuous integration», en *2008 12th European Conference on Software Maintenance and Reengineering*, 2008, pp. 233–242.
- [41] C. F. Carroll, «IT Success and Failure—the Standish Group CHAOS Report Success Factors», *Chris F Carroll*, 2013.
- [42] H. H. Olsson y J. Bosch, «Climbing the “Stairway to Heaven”: evolving from agile development to continuous deployment of software», en *Continuous software engineering*, Springer, 2014, pp. 15–27.
- [43] M. Fowler, «Software Development in the 21st century», presentado en ThoughtWorks XCONF 2014, Hamburg & Manchester, 2014.
- [44] M. Fowler, «Continuous Delivery», *martinfowler.com*. <https://martinfowler.com/bliki/ContinuousDelivery.html> (accedido ene. 20, 2020).
- [45] «Continuous Delivery, Deployment & Integration: 20 Key Differences», *Stackify*, jul. 25, 2017. <https://stackify.com/continuous-delivery-vs-continuous-deployment-vs-continuous-integration/> (accedido ene. 20, 2020).
- [46] M. Virmani, «Understanding DevOps & bridging the gap from continuous integration to continuous delivery», en *Fifth International Conference on the Innovative Computing Technology (INTECH 2015)*, 2015, pp. 78–82.
- [47] A. Phillips, M. Sens, A. De Jonge, y M. Van Holsteijn, «The IT Manager’s Guide to Continuous Delivery: Delivering business value in hours, not months», *XebiaLabs*, 2015.
- [48] M. Vesikivi, «Release management process in a software service company», jun. 2016. [En línea]. Disponible en: <https://aaltodoc.aalto.fi:443/handle/123456789/20972>. (accedido: mar. 31, 2020)
- [49] L. Crispin y J. Gregory, *Agile testing: A practical guide for testers and agile teams*. Pearson Education, 2009.
- [50] L. Crispin y J. Gregory, *More Agile Testing: Learning Journeys for the Whole Team*. Addison-Wesley, 2014.
- [51] «ISO 25010». <http://iso25000.com/index.php/en/iso-25000-standards/iso-25010> (accedido feb. 01, 2020).

- [52] «ISO 25012». <http://iso25000.com/index.php/en/iso-25000-standards/iso-25012> (accedido feb. 01, 2020).
- [53] «ISO 25040». <http://iso25000.com/index.php/en/iso-25000-standards/iso-25040> (accedido feb. 01, 2020).
- [54] D. Graham, E. van Veenendaal, I. Evans, y R. Black, *Foundations of Software Testing: ISTQB Certification*. Int. Thomson Business Press, 2006.
- [55] R. Black *et al.*, «Agile testing foundations: An istqb foundation level agile tester guide», 2017.
- [56] P. Louridas, «Static code analysis», *Ieee Softw.*, vol. 23, n.º 4, pp. 58–61, 2006.
- [57] 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, 1995.
- [58] E. Irrazábal, «Mejora de la mantenibilidad con un modelo de medición de la calidad: resultados en una gran empresa», en *XXI Congreso Argentino de Ciencias de la Computación (Junín, 2015)*, 2015.
- [59] 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*, 2016, pp. 1347–1349.
- [60] «1008-1987 - IEEE Standard for Software Unit Testing». <https://standards.ieee.org/standard/1008-1987.html> (accedido feb. 01, 2020).
- [61] C. Jöngren, *Automated integration testing: An evaluation of cruisecontrol.net*. Skolan för datavetenskap och kommunikation, Kungliga Tekniska högskolan, 2008.
- [62] «ISO/IEC 19501:2005», *ISO*. <http://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/03/26/32620.html> (accedido feb. 03, 2020).
- [63] 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.
- [64] M. T. Nygard, *Release it!: design and deploy production-ready software*. Pragmatic Bookshelf, 2018.
- [65] J. Bach, «Exploratory Testing», *Satisfice, Inc.*. <https://www.satisfice.com/exploratory-testing> (accedido feb. 03, 2020).
- [66] J. A. Whittaker, *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. Pearson Education, 2009.
- [67] «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 feb. 03, 2020).
- [68] M. A. Mascheroni, C. L. Greiner, G. N. Dapozo, y M. G. Estayno, «Herramienta para automatizar la evaluación de la usabilidad en productos software», en *XVIII Congreso Argentino de Ciencias de la Computación*, 2012.
- [69] M. A. Mascheroni, C. L. Greiner, G. N. Dapozo, y M. G. Estayno, «Automatización de la evaluación de la Usabilidad del Software», en *XV Workshop de Investigadores en Ciencias de la Computación*, 2013.
- [70] M. A. Mascheroni, C. L. Greiner, R. H. Petris, G. N. Dapozo, y M. G. Estayno, «Calidad de software e ingeniería de usabilidad», en *XIV*

- Workshop de Investigadores en Ciencias de la Computación*, 2012.
- [71] M. Cohn, *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.
- [72] A. Pope, *Essay on criticism*. JM Dent, 1903.
- [73] V. E. Jyothi, R. R. Krishna, y K. N. Rao, «Agility in Web Application Development–Success Embraces at the Enterprise Level», p. 5.
- [74] 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, 2004.
- [75] J.-L. Lions *et al.*, *Ariane 5 flight 501 failure report by the inquiry board*. european space agency Paris, 1996.
- [76] «List of software bugs», *Wikipedia*. nov. 17, 2019. [En línea]. Disponible en: https://en.wikipedia.org/w/index.php?title=List_of_software_bugs&oldid=926552432. (accedido: feb. 04, 2020)
- [77] M. Hotle, «Agile Development: What's Still Facts and What's Still Fiction?», *Gart. Res. G00175807*, 2010.
- [78] S. Jain y H. Joshi, «Impact of early testing on cost, reliability and release time», en *2016 5th International Conference on Reliability, Infocom Technologies and Optimization (Trends and Future Directions)(ICRITO)*, 2016, pp. 318–322.
- [79] 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. [En línea]. Disponible en: <http://hdl.handle.net/10915/57081>. (accedido: may 20, 2017)
- [80] 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, pp. 1009-1038, sep. 2018, doi: 10.13053/cys-22-3-2794.
- [81] «Continuous Testing: The Missing Link in the Continuous Delivery Process | BlazeMeter», *Continuous Testing: The Missing Link in the Continuous Delivery Process | BlazeMeter*. <https://www.blazemeter.com/blog/continuous-testing-missing-link-continuous-delivery-process> (accedido feb. 25, 2020).
- [82] BlazeMetter, «Continuous Testing in practice. Completing the Continuous Delivery Process». <https://info.blazemeter.com/shift-left-testing> (accedido feb. 25, 2020).
- [83] B. A. Kitchenham, T. Dyba, y M. Jorgensen, «Evidence-based software engineering», en *Proceedings. 26th International Conference on Software Engineering*, 2004, pp. 273–281.
- [84] B. Kitchenham *et al.*, «Systematic literature reviews in software engineering—a tertiary study», *Inf. Softw. Technol.*, vol. 52, n.º 8, pp. 792–805, 2010.
- [85] S. Imtiaz, M. Bano, N. Ikram, y M. Niazi, «A tertiary study: experiences of conducting systematic literature reviews in software engineering», en *Proceedings of the 17th International Conference on Evaluation and Assessment in Software Engineering*, 2013, pp. 177–182.
- [86] K. Beck, «Embracing change with extreme programming», *Computer*, vol. 32, n.º 10, pp. 70-77, 1999, doi: 10.1109/2.796139.
- [87] D. Ståhl y J. Bosch, «Modeling continuous integration practice differences in industry software development», *J. Syst. Softw.*, vol. 87, n.º 1, pp. 48-59,

- 2014, doi: 10.1016/j.jss.2013.08.032.
- [88] E. Collins, V. F. Vicente, y A. De Lucena Jr., «Strategies for Agile software testing automation: An industrial experience», presentado en Proceedings - International Computer Software and Applications Conference, 2012, pp. 440-445, doi: 10.1109/COMPSACW.2012.84.
 - [89] G. de Vries, B. Visser, L. Wilhelmus, A. van Ewijk, M. van Oosterwijk, y B. Linker, «TPI Next-Business Driven Test Process Improvement"», UTN, 2009.
 - [90] B. A. Kitchenham, *Kitchenham, B.: Guidelines for performing Systematic Literature Reviews in software engineering. EBSE Technical Report EBSE-2007-01.* 2007.
 - [91] «Resultado de las búsquedas», https://drive.google.com/open?id=1FATanEUK4eB8_wxTHqLGSoxMdNIB12qd.
 - [92] «Centre for Reviews and Dissemination - Centre for Reviews and Dissemination, The University of York». <https://www.york.ac.uk/crd/> (accedido oct. 14, 2018).
 - [93] K. Wiklund, S. Eldh, D. Sundmark, y K. Lundqvist, «Impediments for software test automation: A systematic literature review», *Softw. Test. Verification Reliab.*, vol. 27, n.º 8, 2017, doi: 10.1002/stvr.1639.
 - [94] T. Martensson, D. Stahl, y J. Bosch, «The EMFIS model - Enable more frequent integration of software», presentado en Proceedings - 43rd Euromicro Conference on Software Engineering and Advanced Applications, SEAA 2017, 2017, pp. 10-17, doi: 10.1109/SEAA.2017.31.
 - [95] A. C. B. G. da Silva, G. F. Carneiro, F. Brito e Abreu, y M. P. Monteiro, «Frequent releases in open source software: A systematic review», *Inf. Switz.*, vol. 8, n.º 3, 2017, doi: 10.3390/info8030109.
 - [96] P. Raulamo-Jurvanen, M. Mäntylä, y V. Garousi, «Choosing the right test automation tool: A grey literature review of practitioner sources», presentado en ACM International Conference Proceeding Series, 2017, vol. Part F128635, pp. 21-30, doi: 10.1145/3084226.3084252.
 - [97] T. Karvonen, W. Behutiye, M. Oivo, y P. Kuvaja, «Systematic literature review on the impacts of agile release engineering practices», *Inf. Softw. Technol.*, vol. 86, pp. 87-100, 2017, doi: 10.1016/j.infsof.2017.01.009.
 - [98] M. E. Serna, M. R. Martínez, y O. P. A. Tamayo, «A model for determining the maturity of automation of software testing as a research and development area», *Comput. Sist.*, vol. 21, n.º 2, pp. 337-352, 2017, doi: 10.13053/CyS-21-2-2723.
 - [99] V. Garousi y M. V. Mäntylä, «When and what to automate in software testing? A multi-vocal literature review», *Inf. Softw. Technol.*, vol. 76, pp. 92-117, ago. 2016, doi: 10.1016/j.infsof.2016.04.015.
 - [100] E. Laukkanen y M. Mantyla, «Build Waiting Time in Continuous Integration- An Initial Interdisciplinary Literature Review», presentado en Proceedings - 2nd International Workshop on Rapid Continuous Software Engineering, RCoSE 2015, 2015, pp. 1-4, doi: 10.1109/RCoSE.2015.8.
 - [101] F. Häser, M. Felderer, y R. Breu, «Software paradigms, assessment types and non-functional requirements in model-based integration testing: A systematic literature review», presentado en ACM International Conference Proceeding Series, 2014, doi: 10.1145/2601248.2601257.
 - [102] D. M. Rafi, K. R. K. Moses, K. Petersen, y M. V. Mäntylä, «Benefits and

- limitations of automated software testing: Systematic literature review and practitioner survey», presentado en 2012 7th International Workshop on Automation of Software Test, AST 2012 - Proceedings, 2012, pp. 36-42, doi: 10.1109/IWAST.2012.6228988.
- [103] Consorcio SIU, «Informe de Gestión 1996-2013 Consorcio SIU». [En línea]. Disponible en: http://www.siu.edu.ar/siu_descarga_publica.php?ds=a63b1fc98c883b99e342088cabec8439.
- [104] «Convenio para el traspaso del SIU», *Consejo Interuniversitario Nacional*. <https://www.cin.edu.ar/convenio-para-el-traspaso-del-siu/> (accedido dic. 17, 2018).
- [105] «Convenio con la provincia de Buenos Aires», *Consejo Interuniversitario Nacional*. <https://www.cin.edu.ar/convenio-con-la-provincia-de-buenos-aires/> (accedido dic. 17, 2018).
- [106] M. Gurmendi, L. M., «Los sistemas como herramientas para fortalecer la gestión municipal», presentado en X Seminario de la RED MUNI «Nuevo rol del Estado, nuevo rol de los Municipios», Universidad Nacional de La Matanza, ago. 2009. [En línea]. Disponible en: http://municipios.unq.edu.ar/modules/mislibros/archivos/Gurmendi_MenendezREDMuni2009.pdf. (accedido: dic. 17, 2018)
- [107] Gurmendi, L., «Creación de comunidades de práctica a través de TICs: la experiencia del SIU», presentado en Tercer Congreso Argentino de Administración Pública: Sociedad, Estado y Administración, San Miguel de Tucumán, jun. 2005. [En línea]. Disponible en: https://aaeap.org.ar/wp-content/uploads/2018/02/Gurmendi_Lujan.pdf. (accedido: dic. 18, 2018)
- [108] E. Wenger y E. Wenger, «Communities of practice: A brief introduction», 2004.
- [109] E. Kaufman y M. Gurmendi, «Comunidades y redes en la innovación: software y back office. El caso de los comites del SIU en la Argentina», *Informática En Argent. Desafíos Espec. Ya Compet.*, 2006.
- [110] M. de L. Gurmendi y R. D. Williams, «La construcción de comunidades de práctica para asegurar el avance de las TICs en el sistema universitario nacional argentino», 2014.
- [111] «Framework SIU-Toba», *Documentación de SIU-Toba*. <http://toba.siu.edu.ar> (accedido feb. 08, 2020).
- [112] M. Menéndez y M. de Gurmendi, «Sistemas para la toma de decisiones en el ámbito universitario», *Coord. Sist. Para Toma Decis. SIU Dir. Ejecutiva–Consorcio SIU*, 2012.
- [113] G. Trutner, «Sistema de información universitaria 2016: metodología de trabajo SIU», 2016.
- [114] «SIU-Guarani/roles - SIU». <http://documentacion.siu.edu.ar/wiki/SIU-Guarani/roles> (accedido feb. 06, 2020).
- [115] «Listado de implementaciones». <https://www.siu.edu.ar/listado-de-implementaciones/> (accedido feb. 06, 2020).
- [116] Y. B. Leau, W. K. Loo, W. Y. Tham, y S. F. Tan, «Software Development Life Cycle AGILE vs Traditional Approaches», p. 6.
- [117] «Documentación - SIU». <http://documentacion.siu.edu.ar> (accedido feb. 07, 2020).
- [118] M. de L. Gurmendi, G. Trutner, y M. Deganis, «SIU-Diaguaita: sistema de gestión de compras, contrataciones y patrimonio», presentado en Simposio

- de Informática en el Estado (SIE 2015) - JAIIO 44 (Rosario, 2015), sep. 2015. [En línea]. Disponible en: <http://sedici.unlp.edu.ar/handle/10915/56052>. (accedido: feb. 07, 2020)
- [119] T. Preston-Werner, «Versionado Semántico 2.0.0-rc.2», *Semantic Versioning*. <https://semver.org/lang/es/> (accedido mar. 28, 2020).
- [120] V. H. M. Domínguez y M. E. C. Bolaños, «SPEM: Software Process Engineering Metamodel», *Rev. Latinoam. Ing. Softw.*, vol. 3, n.º 2, pp. 92-100, 2015, doi: 10.18294/relais.2015.92-100.
- [121] S. OMG y O. Notation, «Software & systems process engineering meta-model specification», *OMG Std Rev*, vol. 2, pp. 18–71, 2008.
- [122] C. Anderson, «Docker [Software engineering]», *IEEE Softw.*, vol. 32, n.º 3, pp. 102-c3, may 2015, doi: 10.1109/MS.2015.62.
- [123] «What is a Container? | App Containerization | Docker». <https://www.docker.com/resources/what-container> (accedido mar. 16, 2020).
- [124] F. Ruiz y J. Verdugo, *Guía de Uso de SPEM 2 con EPF Composer*. 2008.
- [125] E. J. H. Westby, *Git for teams: a user-centered approach to creating efficient workflows in Git*. O'Reilly Media, Inc., 2015.
- [126] «A successful Git branching model», *nvie.com*. <http://nvie.com/posts/a-successful-git-branching-model/> (accedido mar. 26, 2020).
- [127] «SIU-Arai/Arquitectura - SIU». <http://documentacion.siu.edu.ar/wiki/SIU-Arai/Arquitectura> (accedido abr. 09, 2020).
- [128] «Configuration Management - Continuous Delivery». <https://continuousdelivery.com/foundations/configuration-management/> (accedido abr. 10, 2020).
- [129] 1320 Ridder Park Drive, «‘Shift Left’ Approach to Testing to Accelerate and Improve Application Development - Gartner Report». <https://info.blazemeter.com/gartner-report-shift-left-approach-to-testing-to-accelerate-and-improve-application-development-2019-07> (accedido may 24, 2020).
- [130] «IDC Innovators: Tools Supporting Open (Unopinionated) Developer Platforms, 2019», *IDC: The premier global market intelligence company*. <https://www.idc.com/getdoc.jsp?containerId=US45074219> (accedido abr. 04, 2020).
- [131] M. S. Arefeen y M. Schiller, «Continuous Integration Using Gitlab», *Undergrad. Res. Nat. Clin. Sci. Technol. J.*, vol. 3, pp. 1-6, sep. 2019, doi: 10.26685/urncst.152.
- [132] «Modernize Your CI/CD eBook», *Gitlab*. learn.gitlab.com (accedido abr. 05, 2020).

7

ANEXOS

7 ANEXOS

Este capítulo presenta en detalle la descripción de las tecnologías adoptadas para el despliegue de aplicaciones basado en contenedores, así como el uso de Gitlab como la plataforma para CI/CD adoptada.

7.1 Docker para empaquetar aplicaciones

A continuación, se describe la tecnología de contenedores basada en Docker, para empaquetar aplicaciones y su posterior despliegue y orquestación. Finalmente se muestra un ejemplo de cómo se construye la imagen Docker para el proyecto **SIU-Araí: Usuarios**.

7.1.1 Contenedor de una aplicación

La tecnología de contenedores, también conocida simplemente como un contenedor, es una unidad de software estandarizada. Ofrece un método para empaquetar y encapsular una aplicación, con todas sus dependencias, de modo que pueda ser ejecutada en forma aislada de cualquier otro proceso. Este contenedor incluye todo lo necesario para ejecutar la aplicación: código, rutinas de ejecución, herramientas de sistema, librerías y configuración.

Un *contenedor* ofrece muchas ventajas, debido a que el desarrollador de software es quien puede definir el ambiente de ejecución y las dependencias de la aplicación. Luego esta aplicación *contenerizada* se puede ejecutar tanto en otra PC como en un centro de datos o una nube de servicios, sin preocuparse de los requerimientos de la aplicación o las diferencias que puedan existir en los diferentes entornos donde se ejecutase (sistema operativo, librerías, red, software de base).

Docker⁴⁹ es una tecnología que permite empaquetar una aplicación en un formato estándar, también conocido como *imagen docker*. Una *imagen docker* es entonces el resultado de empaquetar una aplicación para su posterior ejecución

49 Docker, <https://www.docker.com/>

como una aplicación *contenerizada*. Docker, además, permite tomar una imagen docker y ponerla en ejecución, generando así una instancia de dicha aplicación. Esto último es lo que se conoce como *contenedor*.

7.1.2 Orquestación de contenedores

La orquestación de contenedores es todo lo relacionado con la administración del ciclo de vida de los contenedores, especialmente en entornos grandes y dinámicos. El uso de la orquestación de contenedores permite controlar y automatizar diversas tareas:

- Aprovisionamiento y despliegue de contenedores
- Redundancia y disponibilidad de contenedores
- Escalado de contenedores, para distribuir la carga de la aplicación a través de toda la infraestructura
- Traslado de contenedores de un nodo a otro en situaciones de escasez de recursos o de inconvenientes técnicos del mismo
- Asignación de recursos entre contenedores
- Exponer hacia el mundo exterior servicios ejecutándose en un contenedor
- Balanceo de carga de servicios entre contenedores
- Monitoreo de salud de los contenedores y nodos
- Configuración de una aplicación en relación a los contenedores

Debido a que el formato de contenedores es un estándar⁵⁰, existe un ecosistema de herramientas y plataformas encargadas de facilitar la tarea de despliegue de este tipo de software en multitud de ambientes. Existen plataformas de nube de servicios públicas, accesibles como servicios, tales como Amazon Web Services (AWS) o Google Cloud Platform (GPC). También existen plataformas para despliegue a demanda en un centro de datos privado, como Rancher, Red Hat OpenShift, Docker Swarm, entre otros. Estas herramientas pueden tener motores de ejecución de contenedores basados en Kubernetes o Docker Engine.

50 OpenContainer es una iniciativa de la LinuxFoundation, encargada de la gobernanza de la especificación del formato de contenedores y su ejecución. <https://www.opencontainers.org/>

Cuando se utiliza una herramienta para la orquestación de contenedores, típicamente se describe la configuración de la aplicación en un archivo de formato YAML. Estos archivos de configuración permiten indicar a la herramienta de orquestación una serie de órdenes: de donde obtener las imágenes de los contenedores, como establecer la red entre contenedores, donde guardar archivos o de donde leer configuraciones para un contenedor en particular, entre otros. Típicamente, esta configuración se versiona como parte de un proceso de gestión de configuración, considerando diferentes entornos donde podrá ser desplegado (desarrollo, pruebas, pre-producción, producción).

7.1.3 Construcción de imagen Docker

Las imágenes Docker se construyen a partir de su definición mediante un archivo *Dockerfile*⁵¹. Este archivo contiene un conjunto de instrucciones que permiten armar de forma repetible y reproducible dicha imagen. A continuación se muestra el contenido del archivo *Dockerfile* para el proyecto SIU-Araí: Usuarios. Este se encuentra dividido en fases, que facilitan la construcción por etapas de cada uno de los módulos que componen el proyecto.

51 Dockerfile, <https://docs.docker.com/engine/reference/builder/>

base	<pre>FROM siutoba/php:7.3-alpine as base RUN apk --no-cache add \ git yarn WORKDIR /usr/local/build RUN curl -sS "https://getcomposer.org/" -o /bin/composer RUN chmod +x /bin/composer RUN echo "composer optimizado" \ && composer config -g cache-dir \$COMPOSER_CACHE_DIR \ && composer config -g platform.php 7.3 \ && composer global require hirak/prestissimo RUN echo "preparar el proyecto" \ && mkdir /usr/local/app/resources/img/foto -p \ && mkdir /usr/local/app/resources/img/iconos -p</pre>
-------------	--

src-idm	<pre>FROM base as src-idm COPY docker /usr/local/build/docker COPY core /usr/local/build/core COPY version /usr/local/build/version COPY idm/composer.json /usr/local/build/idm/composer.json COPY idm/composer.lock /usr/local/build/idm/composer.lock RUN composer install -d ./idm --no-dev --ignore-platform-reqs --prefer-dist --no-progress -n --no-autoloader --no-scripts COPY idm /usr/local/build/idm RUN echo "composer autoload" \ && composer dump-autoload -d ./idm --optimize --apcu \ && composer run-script post-install-cmd -d ./idm RUN echo "preparar el proyecto" \ && cp -R ./idm/templates/app/ idm/instalacion \ && cp -R ./idm/templates/app/www idm/www/ \ && cp ./idm/templates/.env.dist idm/.env \ && echo > ./idm/arai-usuarios.log RUN echo "eliminar archivos innecesarios en imagen" \ && rm -R ./idm/vendor/siu-toba/framework/proyectos \ && rm -R ./idm/vendor/siu-toba/framework/www/swagger \ && find . -type d -name ".git" xargs rm -rf RUN bash ./docker/entrypoint.sh --fix-permissions-idm RUN rm -Rf ./idm/.env</pre>
----------------	--

src-idp	<pre> FROM base as src-idp RUN echo "descarga template login" \ && wget https://url/template-login-master.tar.gz && tar -xzf /tmp/template.tar.gz --directory /tmp RUN echo "deps template login" \ && composer install --working-dir /tmp/template-login COPY docker /usr/local/build/docker COPY core /usr/local/build/core COPY version /usr/local/build/version COPY idp/simplesamlphp /usr/local/build/idp/simplesamlphp COPY idp/composer.json /usr/local/build/idp/composer.json COPY idp/composer.lock /usr/local/build/idp/composer.lock RUN composer install -d ./idp --no-dev --ignore-platform-reqs --prefer-dist --no-progress -n --no-autoloader --no-scripts COPY idp /usr/local/build/idp RUN echo "composer autoload" \ && composer dump-autoload -d ./idp --optimize --apcu RUN echo "descargar dependencias JS de simplesamlphp" \ && apk --no-cache add npm \ && cd idp/vendor/simplesamlphp/simplesamlphp \ && npm install \ && npm run build RUN echo "preparar el proyecto" \ && echo > ./idp/arai-usuarios.log \ && cp -R /tmp/template idp/config/template-login \ && cp ./idp/templates/.env.dist idp/.env \ && cp -R ./idp/templates/app/config/* idp/config RUN echo "eliminar archivos innecesarios de la imagen" \ && find . -type d -name ".idea" xargs rm -rf \ && find . -type d -name ".git" xargs rm -rf RUN bash ./docker/entrypoint.sh --fix-permissions-idp RUN rm -Rf ./idp/.env </pre>
----------------	--

idm	<pre>FROM siutoba/php:7.3-alpine as idm COPY --from=src-idm --chown=apache:apache /usr/local/build /usr/local/app RUN echo "ajustes de apache" \ && sed -i 's/^Listen 80\$/Listen 0.0.0.0:80/' httpd.conf \ && cp templates/vhost.conf /etc/apache2/conf.d/vhost.conf RUN rm -Rf composer-cache</pre>
------------	--

idp	<pre>FROM siutoba/php:7.3-alpine as idp COPY --from=src-idp --chown=apache:apache /usr/local/build /usr/local/app RUN echo "ajustes de apache" \ && sed -i 's/^Listen 80\$/Listen 0.0.0.0:80/' httpd.conf \ && cp templates/vhost.conf /etc/apache2/conf.d/vhost.conf RUN rm -Rf composer-cache</pre>
------------	--

all	<pre>FROM siutoba/php:7.3-alpine as all COPY --from=src-idm --chown=apache:apache /usr/local/build /usr/local/app COPY --from=src-idp --chown=apache:apache /usr/local/build/idp /usr/local/app/idp RUN echo "ajustes de apache" \ && sed -i 's/^Listen 80\$/Listen 0.0.0.0:80/' httpd.conf \ && cp vhost-all.conf /etc/apache2/conf.d/vhost.conf RUN rm -Rf composer-cache</pre>
------------	--

final	<pre>FROM \${APP} as final MAINTAINER arai@siu.edu.ar RUN apk --no-cache add \ php7-ldap \ php7-memcached WORKDIR /usr/local/app COPY ./docker/entrypoint.sh / RUN chmod +x /entrypoint.sh EXPOSE 80 ENTRYPOINT ["/entrypoint.sh"] CMD ["--serve"]</pre>
--------------	--

Con el archivo *Dockerfile* definido previamente, la forma de construir la imagen es ejecutando el siguiente comando

```
docker build -f Dockerfile --build-arg <etapa> .
```

Este comando acepta como argumento la etapa, por lo que genera una imagen Docker para el módulo en cuestión (idp, idm, api).

7.2 Gitlab: plataforma para CI/CD

Gitlab⁵² un proyecto open source, una herramienta relativamente nueva para el soporte de los equipos de desarrollo de software. Git⁵³ es una herramienta popular entre desarrolladores, usada para hacer control de versiones y Gitlab

52 Gitlab, <https://about.gitlab.com>

53 Git, <https://git-scm.com>

proveer repositorios Git, pudiendo además integrarse como servidor de CI/CD⁵⁴. Esto permite automatizar todos los procesos que van desde planificación del proyecto y administración del código fuente, integración y despliegue, seguridad y monitoreo. En el mercado de plataformas de desarrollo abierto, según un reporte del International Data Corporation (IDC), Gitlab se posiciona como una de los cinco innovadores de dicho mercado en 2019 [130].

GitLab administra el ciclo de vida completo del desarrollo de software, permite que un desarrollador tome el código fuente y lo ejecute a través de todos los procesos que sean requeridos, en el ciclo que va desde el desarrollo al despliegue [131] [132]. Para completar todas estas tareas fundamentales en un flujo de CI/CD, muchas plataformas dependen de la integración con otras herramientas externas para alcanzar dichos objetivos. Muchas veces significa costos adicionales para mantener una cadena complicada de diferentes herramientas: conectar un SCM, un servidor de integración, un servidor de artefactos, un servidor de despliegue, un servidor de monitoreo, por mencionar algunos. En vez de sólo enfocarse en construir software, la organización debe mantener y administrar esta cadena de herramientas. En comparación con otras herramientas [132], Gitlab ofrece en una única aplicación⁵⁵ una plataforma consolidada, lo esencial para desplegar un único ambiente de CI/CD, que cubra el ciclo de vida completo del desarrollo y operación de un producto software.

7.2.1 Flujo de trabajo

Git es un sistema de control de versiones distribuido que fue diseñado para ser simple, rápido y soportar el desarrollo no lineal (múltiples ramas de desarrollo en paralelo). El punto fuerte de Git es la capacidad de manipular ramas de desarrollo. Al crear una rama, esta diverge de la línea de desarrollo principal y continúa recibiendo trabajo sin mezclarse con esa línea. En términos de desarrollo de software, significa que es posible tener una rama que contiene código fuente de producción; es posible tener otra rama que contenga el desarrollo de la actual versión; es posible tener muchas otras ramas con características en camino; es posible integrar las ramas a distintas ramas de destino.

54 Gitlab CI/CD, <https://docs.gitlab.com/ee/ci/introduction/>

55 Gitlab como plataforma unificada para CI/CD, <https://about.gitlab.com/ci-cd/>

En cualquier proyecto que dependa de un mecanismo de ramificación del control de versiones, algún tipo de modelo debe ser utilizado para organizar como las ramas serán creadas, cuál será la convención de nombres y como se fusionarán dichas ramas, por mencionar algunas actividades. Un modelo conocido es el llamado GitFlow [126], que consiste en separar cada característica a desarrollar en su propia rama de desarrollo. Una versión simplificada del modelo está compuesta por tres tipos de ramas:

- **master**: rama principal donde el código fuente refleja un estado listo para producción.
- **develop**: rama donde el código fuente refleja un estado con los últimos cambios/características del desarrollo.
- **feature/xyz**: ramas usadas para desarrollar nuevas características para la futura versión a liberar.

El procedimiento de desarrollo es simple. Después que una característica en una rama **feature/xyz** finaliza su desarrollo, es unida o fusionada a la rama **develop**. En este punto, otros desarrolladores pueden sincronizar su trabajo y mantener sus propias ramas de desarrollo actualizadas respecto al sistema. Cuando la rama **develop** alcanza un estado particular (y sea considerado lo suficientemente estable), el encargado de liberar la versión une la rama **develop** a la rama **master**.

7.2.2 Merge Requests

Gitlab soporta peticiones de fusión (o como se lo conoce *merge request*, MR) de forma similar a lo que ofrece Github mediante *pull requests*. Mediante un MR es posible implementar políticas de revisión de código, permitiendo aplicar estas prácticas del desarrollo ágil.

Similar a lo que Humble y Farley [15] recomiendan en su plan de ejecución para cada *commit*, los MR se pueden integrar con un conducto de despliegue o DP. Con ello es posible configurar, por ejemplo, que solo un MR que logre construcciones exitosas pueda ser fusionado. Esto evita el típico problema de “romper la construcción”, que afecta a la capacidad de liberar una versión del

producto y/o su uso en el estado actual de la rama.

7.2.3 Pipeline

Un pipeline es un flujo de actividades que es ejecutado por un servidor de CI/CD. Mediante los pipelines, Gitlab permite implementar integraciones y entregas continuas para el desarrollo continuo del software. En la Figura 43 se muestra el flujo de acciones que puede componer un *pipeline*, como Gitlab entiende se pueden suceder.

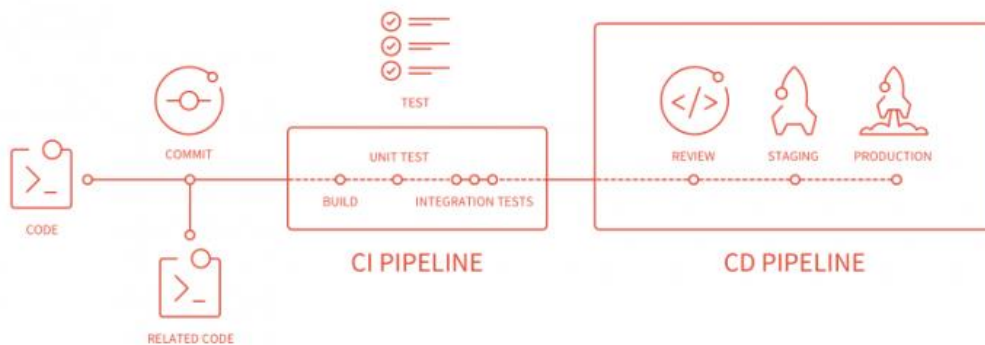


Figura 43 Acciones típicas en un pipeline de Gitlab

Gitlab utiliza el formato YAML⁵⁶ para la definición de dichos *pipelines*. Para agregar un *pipeline* a un proyecto, basta con crear un archivo `.gitlab-ci.yml` en la raíz del repositorio, junto al código fuente. Gitlab lo detecta automáticamente y ejecutará los scripts que tenga definido. Un ejemplo de un *pipeline* sencillo se muestra a continuación:

```
before_script:
  - apt-get install php php-cli -y
run-test:
  script:
    - php --version
```

56 YAML, lenguaje de marcado de texto, de fácil comprensión para el ser humano.

El atributo `before_script` permite especificar cualquier tipo de script que se tenga que ejecutar previamente. El atributo `run-test` define una tarea, permite especificar el script que esta ejecutará. En general, este *pipeline* se ejecutará en cada *commit* del proyecto.