CIMAT Centro de Investigación en Matemáticas, A.C.

ORQUESTACIÓN DE HERRAMIENTAS DE INTEGRACIÓN CONTINUA EN LENGUAJES FUNCIONALES

REPORTE TÉCNICO

Que para obtener el grado de
Maestra en Ingeniería de Software

Presenta
Karina Daniela Chaires López

Director de Reporte Técnico:
Maestro Alejandro García Fernández

Autorización de la versión final

Zacatecas, Zacatecas 7 de febrero del 2019

Agradecimientos

Quiero agradecer en especial a mi familia por su apoyo en cada momento.

Al Mtro. José Guadalupe Hernández Reveles, Coordinador de la Maestría en Ingeniería de Software, por todo el apoyo durante mi estancia en CIMAT, excelente trabajo.

A mi director de reporte técnico, Mtro. Alejandro García Fernández, por sus enseñanzas, tutorías, amistad y orientación. Muchas gracias por compartir todo ese conocimiento.

A mis compañero y amigos que siempre estuvieron allí en todo momento.

A cada uno de los doctores del CIMAT, gracias por compartir sus conocimientos y experiencia.

Al Consejo Nacional de Ciencia y Tecnología (CONACYT) por el financiamiento mediante la beca de posgrado para la realización de mis estudios.

Resumen

La integración continua (IC) y el despliegue continuo (DC), son prácticas con las cuales podemos lograr el aseguramiento de la calidad en los productos de software. Para ello es necesario tener el conocimiento de habilidades, herramientas y procesos de automatización. En los últimos años el resurgimiento del uso del paradigma de programación funcional se debe a que nos permite desarrollar software con alta calidad, en pocas líneas de código, sin efectos secundarios, con datos inmutables, así como programación declarativa y concurrente. Si bien la unión de la Programación Funcional y la Integración Continua debería ser simple, se observó, que no todas las implementaciones en la industria son iguales y en algunos casos ni siquiera se aplica.

El objetivo de este reporte es identificar el **pipeline** (interconexión de varias etapas, las cuales se ejecutan en función de una acción) de herramientas de integración continua utilizadas por la industria y proponer una configuración base de herramientas en lenguajes funcionales.

De las técnicas existentes para llevar a cabo minería en los repositorios de código, se utilizó la técnica de **Mining Software Repositories (MSR)**. Esta técnica selecciona el lenguaje funcional más popular de la industria. A partir de ahí, lleva a cabo un análisis comparativo de los proyectos contenidos en el repositorio de código de GitHub que optan el desarrollo en dicho lenguaje. Se realiza una inspección de los archivos de configuración para determinar cuáles herramientas son las más utilizadas en estos proyectos. Se clasifica cada una de ellas en la etapa de **pipeline** correspondiente al proyecto. Por último se obtiene el compendio de herramientas base utilizadas en cada etapa.

Como resultado, se obtuvo un pipeline de integración continua para lenguajes funcionales. De los 10 proyectos con mayor actividad en la plataforma más popular de repositorios de código, solo 1 utiliza herramientas para las 7 fases del pipeline (Cardano SL).

Podemos concluir que se requiere contar con un pipeline base de IC en lenguajes funcionales, ya que al implementar las herramientas base en cualquier proyecto, asegura la calidad del mismo.

Índice general

Índice de figuras

Índice de tablas

1. Introducción

Las buenas prácticas de software permiten a las organizaciones de la industria contar con entregas frecuentes y la satisfacción del cliente. Como podemos encontrar en la literatura, este tipo de prácticas es de interés, ya que es importante revisar y obtener sistemáticamente los enfoques, herramientas y la documentación de las mismas para su mejor implementación (Shahin et al 2017).

Como parte de la fehaciente necesidad de incorporar mejores prácticas en la industria, se han hecho esfuerzos para poder identificar cuales son las herramientas que las empresas de software necesitan a la hora de proponer sus métodos de automatización y que estos se cumplan.

La reciente investigación realizada por Nicole Forsgren, Jez Humble y Gene Kim durante los años 2016 y 2017, tienen como propósito el identificar acciones claves que toda empresa debe de implementar para poder acelerar su crecimiento de manera exponencial, en esta investigación se observa que el uso de mejores prácticas de software como lo es la **integración continua, entrega continua y contar con la figura de DevOps** en la organización tienen correlación con el desempeño óptimo de la empresa y con ello se puede aumentar la calidad del software y satisfacción del cliente.

Es por ello que la integración continua, como una de las prácticas de extreme programing (Beck 1999), se ha vuelto popular en el desarrollo de software. Se informa que mejora la frecuencia y predictibilidad de la liberación (Goodman y Elbaz 2008), aumenta la productividad del desarrollador (Miller 2008) y mejora la comunicación (Downs et al 2010), entre otros beneficios. Existen grandes diferencias en la medida en que los profesionales en

proyectos de desarrollo de software industrial han experimentado esos beneficios (Ståhl y Bosch 2013).

Por otro lado la Programación Funcional (PF) es un paradigma de programación en el cual la evaluación matemática de funciones es el bloque principal en la construcción del software, debido a esto es que son los más adecuados para manejar el paralelismo y la concurrencia (Khanfor, A., y Yang, Y. 2017).

Se conjetura que el uso de lenguajes funcionales en combinación con prácticas de integración continua ayudan a mejorar la calidad del software e incrementar la satisfacción del cliente (Humble, J., y Kim, G. 2018).

En el presente reporte técnico, se expone la investigación que se realizó para encontrar la orquestación de IC mediante la aplicación de la técnica de Mining Software Repositories (MSR) en los proyectos desarrollados en Haskell, el cual de acuerdo a la literatura y el uso en GitHub y Stack Overflow es el lenguaje funcional más popular.

En el presente trabajo nos planteamos las siguientes preguntas de investigación:
Q1 ¿Se utiliza la integración continua en proyectos de programación funcional?
Q2 ¿Cuáles son las herramientas de integración continua utilizadas en los proyectos?
Q3 ¿Existe un pipeline general para otro tipo de lenguaje, por ejemplo, lenguajes estructurados?
Q4. ¿Se requiere un pipeline de integración continua para proyectos de lenguajes funcionales?

El objetivo general del presente reporte técnico es determinar un pipeline de integración continua para lenguajes de programación funcional basado en el estado de la prácticas.

2. Antecedentes

Integración Continua

En el año de 1991, Grady Booch introduce el concepto de Integración Continua (CI), definiendola como una práctica en la cual "*En intervalos regulares, el proceso de CI produce liberaciones ejecutables que crecen en funcionalidad en cada liberación*" (Hilton et al 2016). En el 2001 se introduce la metodología de Extreme Programing la cual prescribe un número de prácticas técnicas, adoptando a la IC como una de sus prácticas centrales. Dicha práctica comenzó a tomar mayor aceptación cuando Martin Fowler la incluye en sus publicaciones e implementa el primer sistema de IC llamado *CruiseControl*, teniendo como idea principal que cuando más a menudo se integra un proyecto, es mejor y la clave está en la automatización (Humble, J., y Kim, G. 2018), es así que Martin Fowler la define como :

"*La integración continua es una práctica de desarrollo de software donde los miembros de un equipo integran su trabajo con frecuencia, por lo general cada persona se integra al menos diariamente, lo que lleva a integraciones múltiples por día. Cada integración se verifica mediante una compilación automatizada, incluida la prueba, para detectar errores de integración lo más rápido posible. Muchos equipos encuentran que este enfoque conduce a problemas de integración significativamente reducidos y permite a un equipo desarrollar software cohesivo más rápidamente*" (Fowler 2006).

Por otro lado Ståhl, D. la define como:

"*La integración continua, es todo sistema que contenga actividades automatizadas para garantizar que el software pueda ser puesto en producción en cualquier momento, contando con código potencialmente implementable, con calidad, contenido y funcionalidad*" (Ståhl, D., y Bosch, J. 2017).

Entonces para tener una mejor adopción e implementación de las prácticas de IC, es importante realizar un análisis que soporte científicamente los enfoques, herramientas, desafíos y resultados de la implementación de dichas prácticas, y así ayudar a las organizaciones a acelerar su desarrollo y liberación sin dejar de lado la calidad, proporcionando beneficios como (Shahin et al 2017):

- Obtener en menor tiempo la retroalimentación del proceso de desarrollo de software y de los clientes.
- Contar con entregas confiables y en menor tiempo, potenciando la satisfacción del cliente y calidad del producto.
- Fortalecer las relaciones entre los equipos de desarrollo y de despliegue, así como eliminar las tareas manuales.

Es así como las capacidades de integración y entrega continuas son importantes en la industria, invirtiendo recursos para construir las capacidades necesarias, desarrollando soluciones y aplicando estudios para la implementación de IC a gran escala (Ståhl, D., y Bosch, J. 2017), y que los equipos de desarrollo de software pueden reducir el tiempo y re-trabajo que le toma al integrar una tarea en la cual llevan días o semanas trabajando, esto siempre y cuando se aplique el concepto de trabajar en pequeñas tareas, y así lograr construir con calidad e integrar su código dentro del tronco o rama principal del repositorio de código de manera frecuente, ya que cada cambio que se realice construye un proceso que incluye la ejecución de todo el conjunto de pruebas, si alguna parte del proceso falla, los desarrolladores lo pueden arreglar de manera oportuna (Humble, J., y Kim, G. 2018).

Una investigación realizada por Hilton, m et al., en la cual se analizaron 1,529,291 proyectos de código abierto como el objetivo de obtener el uso de la IC en los mismos, en la Fig. 2.1 podemos observar sus resultados, los cuales muestran que el uso de las prácticas de IC a lo largo del tiempo han experimentado un incremento (Hilton, M., Tunnell, T. et al 2016).

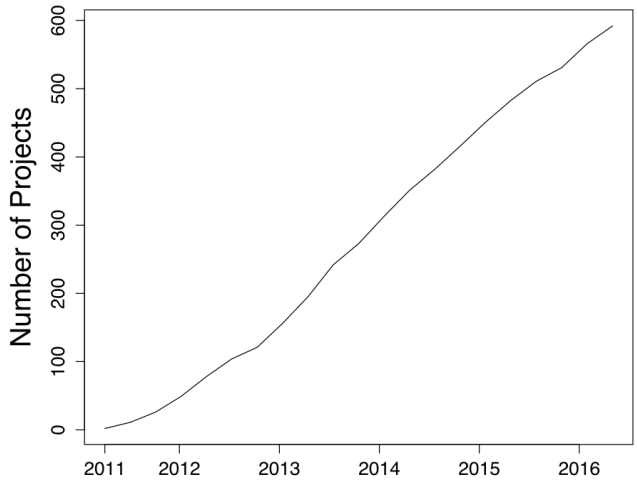

Figura 2.1.*Número de proyectos que utilizan CI a lo largo del tiempo.* Adoptado de "Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016, 426–437. https://doi.org/10.1145/2970276.2970358".

En el desarrollo de software ágil moderno, con las prácticas de IC se propone organizar al equipo en torno a herramientas que permitan integrar y generar versiones de software de manera continua (Bollati et al 2017), estas prácticas son adaptadas por el movimiento de DevOps los cuales las implementan y forman parte principal de la automatización de los procesos de desarrollo de software para gestionar y hacer entregas de manera robusta y con mayor calidad.

Definición y aspectos importantes de un DevOps

Un ingeniero DevOps en su forma más pura es el puente para solventar la brecha que existe entre los desarrolladores de software y los equipos de operación, aumentando la tasa de entrega de software (Michelsen, J. 2014).

De acuerdo a Len Bass, Info Weber y Liming Zhu (2015), en su libro *DevOps*: A Software Architect's Perspective (SEI Series in Software Engineering) definen DevOps como: "*DevOps es un conjunto de prácticas con el propósito de reducir el tiempo entre el hacer un cambio en un sistema y realizar el cambio de manera normal en producción, asegurando una alta calidad*" (Len Bass et al 2015).

Esta definición está orientada a objetivos que nos den la certeza de:

- Contar con mecanismos de entrega de alta calidad, implica alta confiabilidad y repetibilidad en ello, ya que si estos mecanismos presentan fallas con regularidad, el tiempo requerido para realizar la entrega aumenta.
- Identificar las dos principales etapas, la primera, cuando el desarrollador realiza el commit al código y se realizan las pruebas automatizadas, así como el monitoreo del mismo para identificar problemas potenciales. La segunda cuando el código es desplegado en producción.
- Garantizar una implementación de alta calidad a lo largo del ciclo de vida del sistema.

Aunque el rol de *DevOps* no es el mismo en las compañías, existen dos aspectos con los cuales estarán en contacto cotidiano: la automatización y la integración continua (Michelsen, J. 2014).

Automatización: Al automatizar tareas que cotidianamente son realizadas de manera manual, como lo puede ser el respaldo de algún servidor, hace que los ingenieros se enfoquen en tareas más críticas como el solucionar problemas del servidor.

Los ingenieros *DevOps* pueden automatizar los procesos de configuración del servidor creando *scripts* del entorno, estos *scripts* son ejecutados en un nodo y para escalar se requiere hacer mediante la gestión de la configuración mediante herramientas como *Chef*, *Puppet* y *Ansible* (por mencionar algunas) que ayuden a realizar dicha tarea (Michelsen, J. 2014).

Integración continua: Existen diferentes productos y herramientas las cuales ayudan a las organizaciones a implementar el proceso de integración continua, algunas herramientas pueden estar en la propia infraestructura de la compañía, mientras que otros productos como *Travis*, *CircleCI*, *GitLab CI*, *Semaphore*, *AppVeyor* o *Jenkins*, son hospedados en la

nube, los DevOps estarán a cargo de realizar las configuraciones e instalaciones necesarias para que los builds del producto de software se realicen correctamente (Michelsen, J. 2014).

Len Bass et al (2015), identificaron 5 categorías de las prácticas *DevOps*:

1. Tratar al personal de operaciones como el principal usuario desde el punto de vista de los requerimientos: El participar en la etapa de requerimientos, garantizará que los diferentes tipos de requerimientos sean considerados.
2. Hacer que el equipo de desarrollo sea más responsable del manejo de incidentes relevantes: Estas prácticas son referentes a disminuir el tiempo entre la detección del error y la reparación del mismo.
3. Hacer cumplir el proceso de implementación utilizado por todos, incluido el equipo de desarrollo y de operaciones: Esta práctica busca asegurar el desarrollo de alta calidad y también hace referencia a el tiempo que se toma en analizar y reparar el error.
4. Uso de despliegues continuos: Esta práctica enfatiza la automatización de pruebas que incrementen la calidad del despliegue del código en producción.
5. Realizar la configuración de la infraestructura, como los scripts de implementación, utilizando el mismo conjunto de prácticas que el código de la aplicación.

En la Fig. 2.2 se visualizan los procesos generales de los *DevOps*, donde se presentan las actividades realizadas en cada fase del ciclo de vida, las cuales tienen relación con las principales prácticas descritas anteriormente.

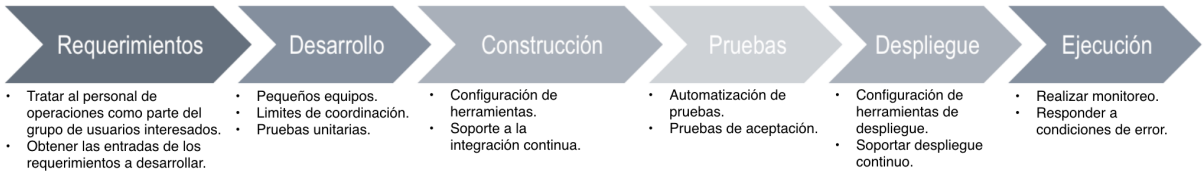

Figura 2.2.*Proceso del ciclo de vida DevOps.* Adoptado de “Len bass, 2015”.

Michelsen, J. (2014) considera que un *DevOps* debe tener dominio sobre aspectos de hardware y de software, ya que son importantes para que los ingenieros logren desarrollar habilidades como:

- Experiencia como *SysAdmin*, al grado de ser un experto.
- Experiencia en virtualización.
- Tener una formación técnica sólida y multidisciplinaria.
- Buena experiencia con los scripts.
- Entrenamiento en seguridad.
- Experiencia en automatización de herramientas.
- Desarrollo de Pruebas.
- Habilidades de trabajo en equipo.
- Buena comunicación.
- Hacer la infraestructura resistente a fallos.

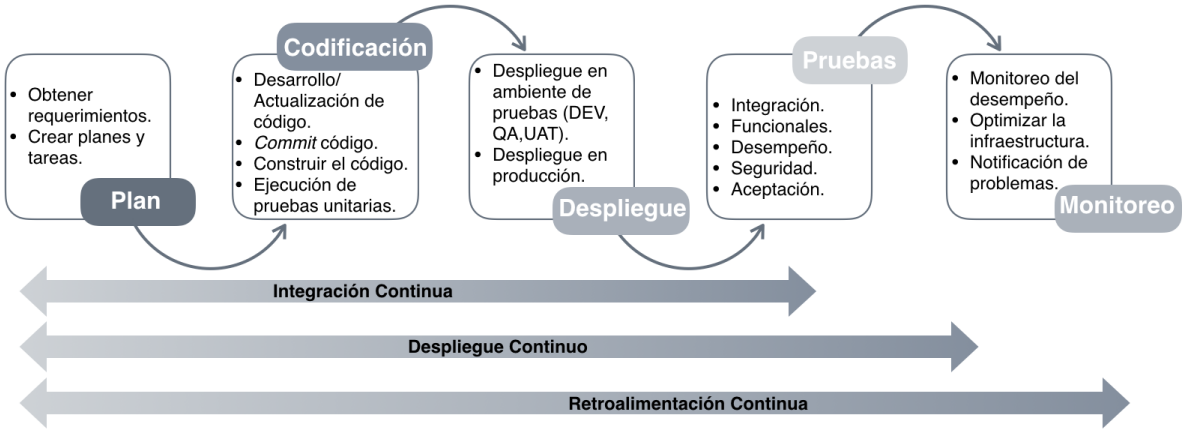

Figura 2.3.*Roles del ingeniero DevOps.* Adoptado de "Michelsen, J. (2014). A Pragmatic Guide to Getting Started with DevOps. CA Technologies, 26. https://doi.org/10.1017/S0031182000064192".

Si bien el porcentaje de personas que reporta trabajar con equipos *DevOps* ha ido incrementando en los últimos años, pasando de un 16% en 2014 a un 27% en el 2017 (Humble, J., y Kim, G. 2018), es importante que las compañías adopten estrategias las cuales respondan a las demandas de los clientes para satisfacer sus necesidades de

calidad en menor tiempo, y para ello es necesario incorporar en los equipos a *DevOps* que coadyuven a trabajar en un ambiente de colaboración rompiendo la brecha que existe entre los desarrolladores de software y los equipos de operación, ya que invertir en ellos tiene alta correlación con el rendimiento en el continuo despliegue de software (Michelsen, J. 2014).

Fases de la integración continua

Para transferir el código desde el repositorio de código al entorno de producción, se deben de incluir etapas las cuales conformarán el pipeline para completar dicha acción (Shahin et al 2017).

De acuerdo al ISTQB (*International Software Testing Qualifications Board)* el ciclo de integración continua para verificar la validez del código es el siguiente:

- **Análisis de código estático:** Proceso automatizado que realiza una revisión de código, con el objetivo de encontrar posibles vulnerabilidades en el mismo, por ejemplo de seguridad (Owasp), así como el cumplimiento de los estándares de código y las mejores prácticas (Eddy, B. P. et al 2017).
- **Compilación**: Es el paso en el que a partir de una entrada de código fuente o un archivo de configuración, se genera un artefacto ejecutable de la aplicación (Len bass 2015).
- **Ejecución de pruebas unitarias:** Se verifica de forma aislada del resto del sistema el correcto comportamiento de las unidades individuales de código, por ejemplo las funciones o métodos, al probar cada unidad de código de manera individual permiten a los desarrolladores identificar fácilmente en dónde ocurre el error (Eddy, B. P. et al 2017); organizada en tres secciones **a) organizar** desarrollar la prueba de unidad, asegurándose que esta cubra toda la funcionalidad a ser probada, **b) actuar** ejecutar la prueba con las diferentes entradas de datos que se pudieran tener y **c) afirmar** verificar que el comportamiento y resultado sea correctos mediante aserciones (Felbinger et al 2018).

- **Implementación:** Generación de código para ser desplegado en un entorno de pruebas o de producción (ISTQB).
- **Ejecución de pruebas de integración:** Es el paso en el que el artefacto ejecutable es probado en ambientes de pruebas que incluyen conexiones con servicios externos (Len bass 2015).
- **Reportes:** Es el panel de control visual en el cual se identifican mediante colores (por ejemplo rojo, verde y amarillo) el estado de los parámetros claves definidos por cada organización (ISTQB).

En la Fig. 2.4 nos muestra cada una de las actividades que conforman la integración continua, de acuerdo al ISTQB (*International Software Testing Qualifications Board).*

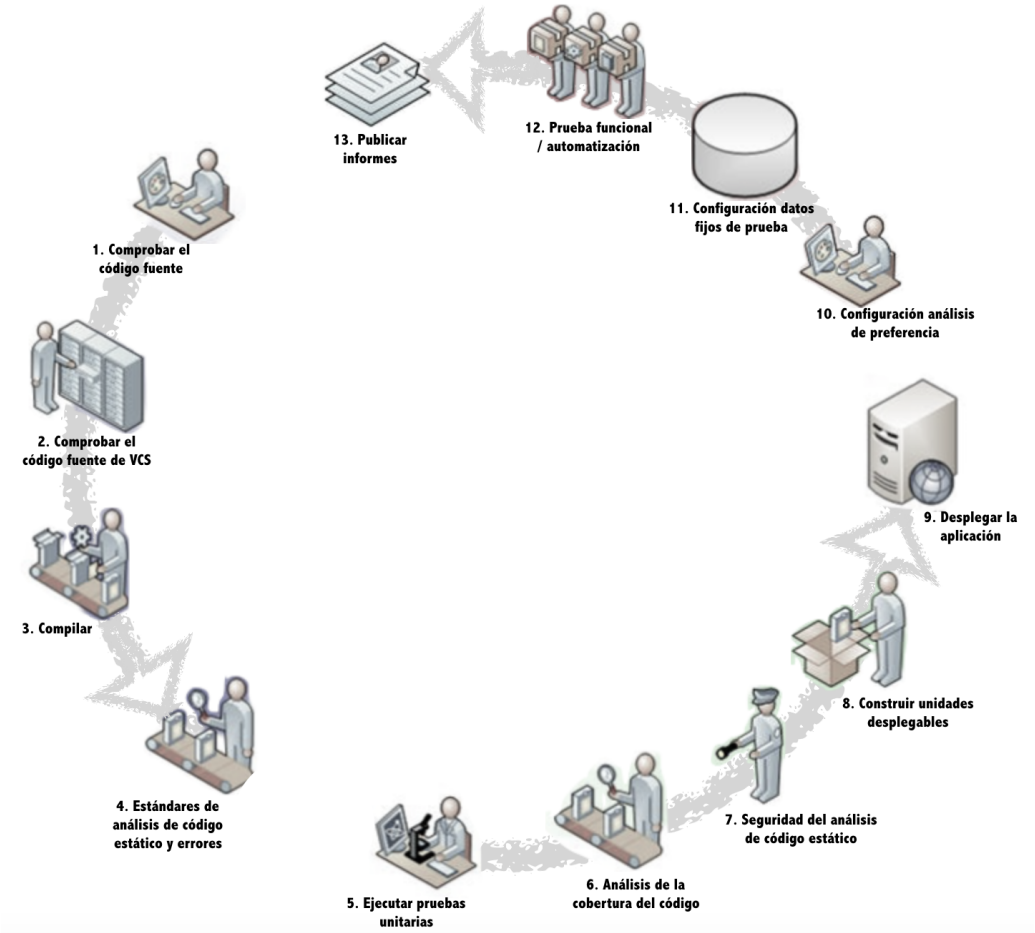

Figura 2.4.*Ciclo de Integración Continua.* Adoptado de "What is Continuous Integration in Agile methodology? (n.d.). Retrieved, from http://istqbexamcertification.com/what-is-continuous-integration-in-agile-methodology/".

Dentro de estas fases es importante recordar que uno de los actores principales son los *DevOps*, Marijan, D. et al (2018), realizaron una investigación dando como resultado la propuesta de una técnica de integración continua, en la cual se realiza una selección de pruebas optimizadas basadas en fallas o riesgos, y así reducir el tiempo en que estas se ejecutan, sin comprometer la calidad.

En la Fig. 2.5 se pueden visualizar las fases que se llevan a cabo en esta técnica:

1. Hacer *commit* para realizar los cambios en el repositorio de código.
2. El servidor de integración compila todos los cambios individuales.
3. Se inician las pruebas.
4. Seleccionar la información que se utilizará para todas las etapas de pruebas, realizando el proceso de optimización, en el cual, de una base de datos, que contiene información histórica de las pruebas, operaciones, casos de pruebas, reglas de negocio que tuvieron mayor cantidad de errores en el pasado, así como los tiempos de ejecución y cobertura de la prueba, el algoritmo de optimización calcula y selecciona el conjunto y orden de las pruebas que se deben de correr, para acelerar y mejorar la calidad de las pruebas de integración y configuración.
5. Ejecución de pruebas de unidad.
6. Ejecución de pruebas de integración.
7. Ejecución de pruebas de configuración.
8. Ejecución de pruebas de desempeño.
9. Ejecución de pruebas de aceptación.
10. Guardar el resultado de las pruebas.
11. Una vez que las pruebas se han ejecutado con éxito, el código se implementa en producción.
12. Monitoreo de la operación.

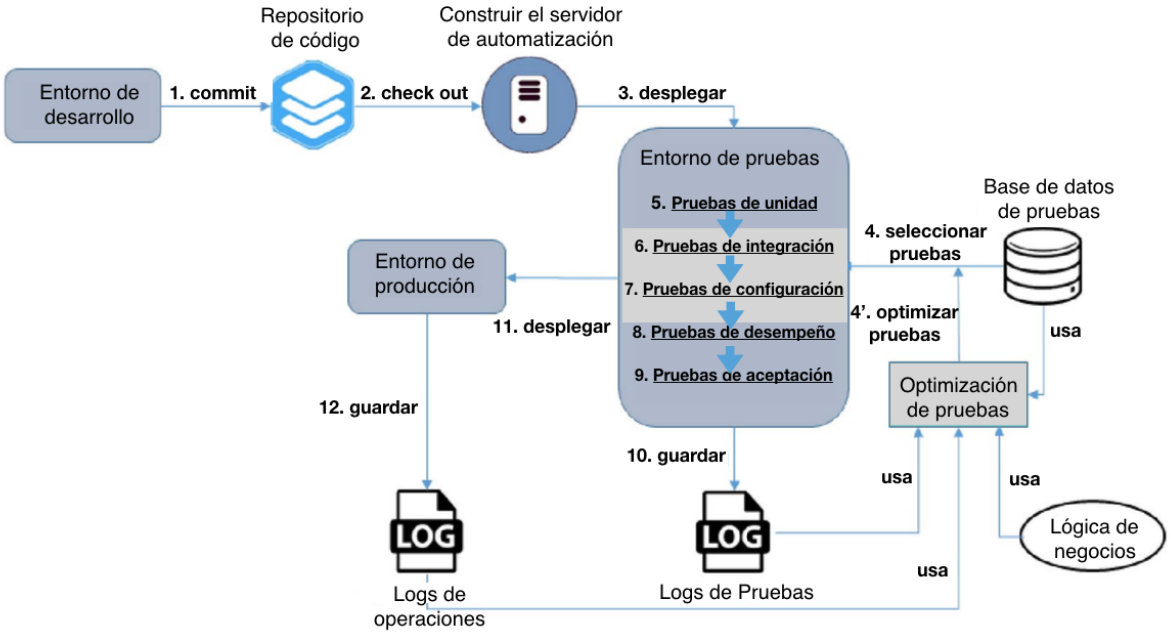

Figura 2.5.*DevOps con optimización de pruebas para CI.* Adoptado de "Marijan, D., Liaaen, M., & Sen, S. (2018). DevOps Improvements for Reduced Cycle Times with Integrated Test Optimizations for Continuous Integration. 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 22–27. https://doi.org/10.1109/COMPSAC.2018.00012".

Por otro lado en la Fig. 2.6, de acuerdo a Eddy, B. P. et al (2017), cuando un desarrollador realiza un cambio y hace commit al repositorio de código, se activa el proceso de integración continua ejecutando las etapas más comunes de este proceso: análisis estático, pruebas unitarias, pruebas de integración, implementación.

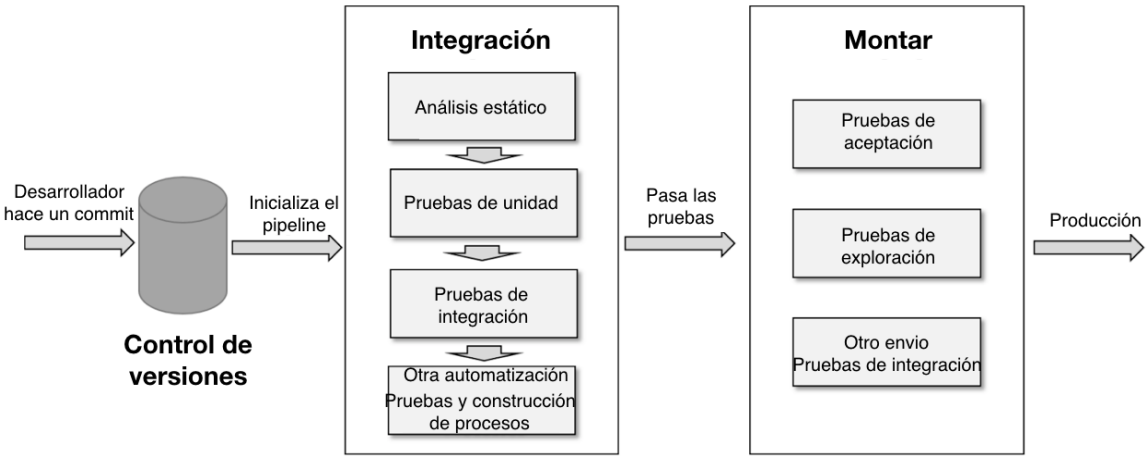

Figura 2.6.*Etapas de una simplificación del Proceso de Integración Continua.* Adoptado de "Eddy, B. P., Wilde, N., Cooper, N. A., Mishra, B., Gamboa, V. S., Shah, K. M., Shields, N. A. (2017). A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses. *Proceedings - 30th IEEE Conference on Software Engineering Education and Training, CSEE and T 2017*, *2017–January*, 47–56. https://doi.org/10.1109/CSEET.2017.18".

Una vez que el código pasa con éxito por estas fases, este es enviado a un ambiente de desarrollo comenzando una las fase de Entrega Continua *(Continuous Delivery)* y Despliegue Continuo (*Continuou*s Deployment) (Eddy, B. P. et al 2017).

¿Qué es la entrega continua y el despliegue continuo?

Como se describió anteriormente, en la integración continua, el principal enfoque es mantener una alta calidad de código así como el reducir el tiempo y esfuerzo que se toman las pruebas. Mientras que para la entrega continua es incorporar pruebas y análisis de la integración continua, así como mejorar la canalización de herramientas para el equipo de control de calidad (QA) en los esquemas de implementación dentro de los entornos de pruebas (Eddy, B. P. et al 2017), esto para garantizar que la aplicación esté siempre lista para pasar a producción, una vez que el código pase con éxito todas las pruebas automatizadas y el control de calidad (Shahin, M. et al 2017).

De acuerdo a Eddy, B. P. et al (2017), la entrega continua emplea prácticas como:

- Un sistema de integración continua para pruebas automatizadas.
- Un sistema de archivos para versionar y poder realizar una regresión de código cuando sea necesario.
- Un proceso de implementación para mover una compilación que pasa en un entorno de pruebas antes de que se apruebe su despliegue en producción.
- Un sistema de notificación para notificar a los desarrolladores si la compilación fue exitosa o fracasó.
- Un sistema de retroalimentación para el equipo de control de calidad (QA) que permite la discusión de cambios que deben realizarse antes de que se envíe a producción.

Por otro lado el despliegue continuo según (Humble, J., y Kim, G. 2018) es “Un conjunto de capacidades que nos permite realizar cambios de todo tipo (características, cambios de

configuración, corrección de errores, experimentos) en producción de manera segura, rápida y sostenible".

Si bien la entrega continua y el despliegue continuo pueden llegar a parecer iguales, la principal diferencia entre ellos, es que el primero tiene un proceso manual, que es la aplicación y aprobación exitosa de pruebas de control de calidad antes de que un cambio de código se envíe a producción (Eddy, B. P. et al 2017), mientras tanto en el despliegue continuo cuando los desarrolladores suben un cambio al repositorio de código, este se ve reflejado en producción automáticamente una vez que pase exitosamente por el canal de implementación (Shahin, M. et al, 2017), y los usuarios finales son los que realizan las pruebas de control de calidad (Eddy, B. P. et al 2017).

Podemos ver que la entrega continua es un enfoque basado en el *pull* en el que la empresa decide qué y cuándo implementar, en el caso del despliegue continuo, esté tiene un enfoque basado en el *push*, es decir la entrega continua no incluye la implementación frecuente y automatizada, por lo tanto el despliegue continuo es consecuente a la entrega continua (Shahin, M. et al 2017).

En la Fig. 2.7, podemos visualizar como es la relación entre estos tres procesos de ingeniería continua de software.

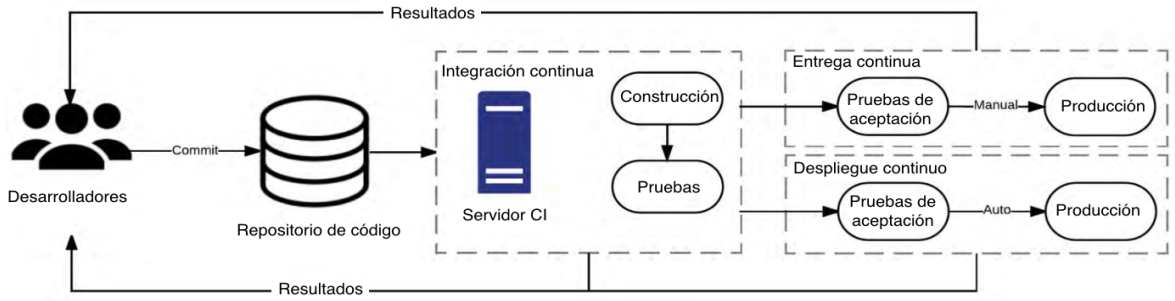

Figura 2.7.*Relación entre integración continua, entrega y despliegue.* Adoptado de "Shahin, M., Babar, M. A. L. I., & Zhu, L. (2017). Continuous Integration , Delivery and Deployment : A Systematic Review on Approaches , Tools , Challenges and Practices, (Ci), 3909–3943./".

Para lograr una automatización de inicio a fin, necesitamos un enfoque que se centre en las herramientas de automatización para adoptar un pipeline de las diferentes tipos de tecnología en cada fase (Lu, H. et al 2017), y así poder utilizarlas en cada una de las prácticas comunes que conllevan estos procesos de integración: un único repositorio de código, construcción automatizada, prácticas de pruebas automáticas, mantener commit simples, construcción rápida, los entornos de pruebas y producción deben ser iguales en sus configuraciones y características, un solo punto de presencia ejecutable, informes y despliegue automático (O. Lavriv et al 2017).

Pipeline

Consiste en una interconexión de varias etapas, las cuales se ejecutan en función de una acción, como lo es hacer commit al código, cada etapa da cumplimiento a múltiples tareas que se ejecutan una a una, además de permitir un incremento de características en cada ciclo (Düllmann, T. et al 2018).

Shahin, M. et al (2017) realizaron una amplia investigación dentro de la literatura para obtener el conjunto de herramientas e infraestructura que componen un pipeline con el objetivo de disminuir la complejidad de adoptar e implementar las prácticas de integración, entrega y despliegue continuo. Dentro de los 69 estudios seleccionados dentro de la literatura, sólo 25 de ellos mostraban el cómo se integran las diferentes herramientas dentro del pipeline, y señalan que como tal no existe un estándar de este.

En la Fig. 2.8, se visualiza la conformación del pipeline el cual lo integran 7 etapas que son:

1. Sistema de control de versiones.
2. Herramienta de análisis y gestión de código.
3. Herramienta de construcción.

4. Servidor de integración continua.
5. Herramienta de prueba.
6. Configuración y aprovisionamiento.
7. Servidor de entrega o despliegue continuo.

Mencionan que no es obligatorio contar con las todas las etapas, ya que dentro de su investigación no encontraron ningún proyecto que implementará todas las etapas.

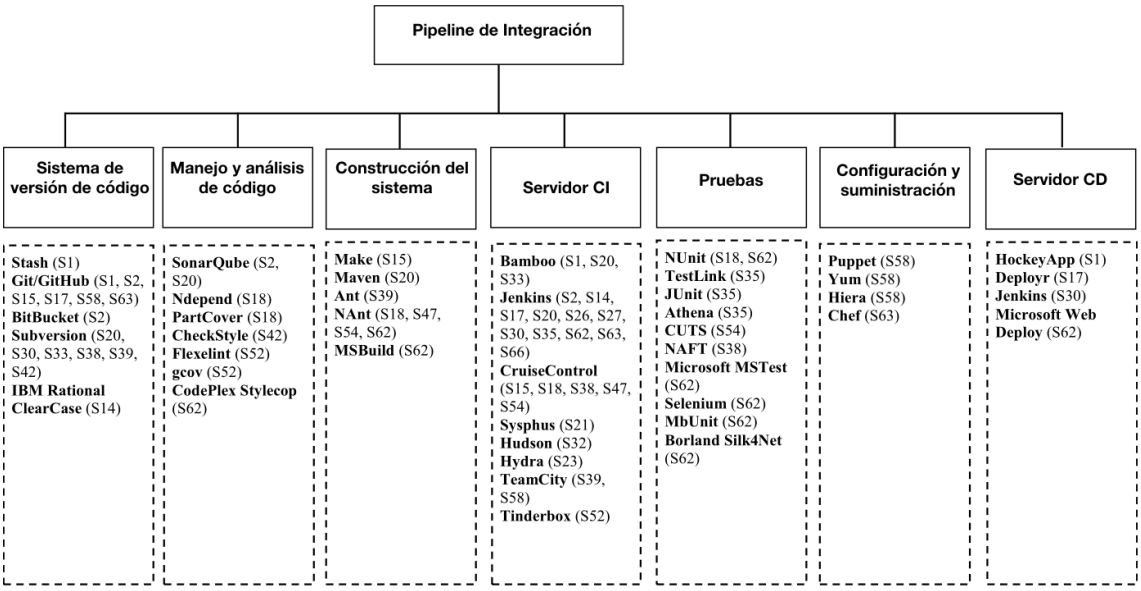

Figura 2.8.*Vista general de las herramientas utilizadas para formar el pipeline de integración.* Adoptado de "Shahin, M., Babar, M. A. L. I., & Zhu, L. (2017). Continuous Integration , Delivery and Deployment : A Systematic Review on Approaches , Tools , Challenges and Practices, (Ci), 3909–3943./".

De manera general el funcionamiento de un pipeline se explica de la siguiente manera: primero se tiene que activar, esto puede ser mediante la acción de descargar el código fuente desde el repositorio de código, este contiene un archivo el cual define la estructura y comportamiento que tendrá el pipeline, por ejemplo un archivo de configuración. Al leer el archivo del pipeline la herramienta encargada del mismo conoce las etapas y tareas que se deben de ejecutar durante la compilación. Posteriormente se compila el código fuente, para asegurarnos de que no existan errores. Para asegurar que la aplicación funciona de manera correcta, se implementan y ejecutan entornos de pruebas, para verificar que no existan problemas en el funcionamiento, una vez se pase estas fases de manera exitosa, la

aplicación está lista para desplegarse en el ambiente de producción. El pipeline puede llegar a tener una infraestructura compleja como lo son servidores alojados en la nube o externos (Düllmann, T. F. et al 2018). El punto de partida para detonar el servidor de integración continua es un sistema de control de versión de código (O. Lavriv et al 2017).

Sistemas de control de versiones

Los sistemas de control de versiones (SCV) son claves en cualquier proyecto de desarrollo, en la actualidad existen varias herramientas como lo es CVS, SVN, Git, BitBucket, GitHub entre otros, los cuales son utilizados ampliamente por los equipos de desarrollo ya que permiten gestionar distintas versiones de código y con ello hace que el proceso de desarrollo sea más productivo (Hu, Y. et al 2016).

Como lo observamos anteriormente en la investigación Shahin, M. et al (2017), GitHub es uno de los repositorios de código abierto mayormente utilizado en todo el mundo, ya que tiene cerca de 9 millones de usuarios y 17 millones de repositorios públicos (Borges, H. et al 2017).

GitHub además provee un API la cual contiene los datos histórico esenciales de los repositorios de código, como lo son el número de estrellas, commits, contribuidores, forks, pull request, entre otros y estos nos sirven para poder obtener aquellos proyectos que requerimos de acuerdo a la investigación que estemos realizando (Borges, H. et al 2017). Estos datos se han utilizado para investigar la colaboración de los usuarios de Github en función de sus actividades en los repositorios, la importancia del lenguaje o predecir las tendencias de los lenguajes de programación más populares (Hu, Y. et al 2016).

De acuerdo a Sajedi Badashian et al (2014), algunas métricas de actividad para cada desarrollador en GitHub son las siguientes:

- ***Commits***: Número de confirmaciones realizadas por el desarrollador.
- ***PullRequets***: la cantidad de veces que el desarrollador notificó a otros sobre sus cambios para que puedan hacerlo si lo desean.
- ***PullReqsHandled***: número de veces que el desarrollador abrió o cerró solicitudes de extracción; la persona que actúa en una solicitud de extracción puede ser diferente de la que emitió esta solicitud de extracción.
- ***ProjectsWatched***: Número de proyectos que el desarrollador observa. Ver un proyecto le permite al usuario ser notificado sobre nuevas confirmaciones, solicitudes de extracción y problemas en el repositorio del proyecto.
- ***IssueComments***: Número de comentarios que el desarrollador hizo sobre los problemas. Los problemas son, en efecto, notificaciones entre los miembros del equipo para el seguimiento de errores o tareas.
- ***IssuesReported***: Número de problemas que informó el desarrollador. Problemas manejados: la cantidad de veces que el desarrollador presionó, bifurcó o comentó sobre un problema previamente informado.
- ***Followers***: Número de personas que siguen al desarrollador.
- ***Mentions***: Número de veces que el nombre del desarrollador se menciona en los comentarios (es decir, comentar comentarios, extraer comentarios de solicitud y emitir comentarios).

Aplicación de la minería de procesos a los repositorios de código de software

La minería de repositorios de código por sus sigla en inglés MSR (Mining software repositories), se enfoca en extraer información de los registros de eventos producidos por un sistema de información (Poncin W. et al 2011) y analizar información de los atributos de

los proyectos de software que se encuentren en los repositorio de código, con ello los investigadores pueden explorar información que ayude a realizar (Siddiqui, T., y Ahmad, A. 2018):

- Una predicción de errores.
- La co-evolución de la producción y código de prueba.
- Un análisis de impacto.
- La predicción del esfuerzo.
- Un análisis de similitud.
- La predicción del cambio arquitectónico del software.
- Inteligencia del software.

La metodología a seguir de acuerdo a Poncin W. et al (2011), es la siguiente:

A) Definir el caso

Es necesario formar una instancia del proceso que puede formarse mediante confirmaciones que tuvieron lugar en una fecha determinada o durante un período determinado (ya sea desde la instancia del desarrollador, un componente, etc.), la técnica de análisis que se aplique, depende de cómo se defina el caso.

B) Extraer el evento

Debido a que no todos los repositorios de código especifican explícitamente los eventos, es necesario especificar la forma para extraer los datos del repositorio. Para cada evento extraído, debemos determinar a qué actividad pertenece el evento, cuándo tuvo lugar el evento y qué datos están asociados con el evento.

C) Relacionar las diferentes representaciones

Diferentes repositorios de software pueden involucrar diferentes representaciones de la misma entidad, por lo tanto, uno debería ser capaz de indicar enlaces entre repositorios: ya sea manual o automáticamente.

¿Qué es la programación funcional?

Antes de definir lo que es la programación funcional, definiremos que es una función, de acuerdo a Aravinth A., Machiraju S. (2018) una función es “Un fragmento de código que puede llamarse por su nombre, se puede utilizar para pasar argumentos sobre los que puede operar y devolver valores opcionalmente”.

Es así que la Programación Funcional (PF) es un paradigma de programación en el cual la evaluación matemática de funciones es el bloque principal en la construcción del software (Khanfor, A., y Yang, Y. 2017). Tomando la definición matemática como esencia, de acuerdo a Aravinth A., Machiraju S. (2018) existen cuatro puntos claves en ella:

1. Una función siempre debe tomar un argumento.
2. Una función siempre debe devolver un valor.
3. Una función debe actuar sólo en sus argumentos de recepción (es decir, X), no en el espacio de nombres global.
4. Para una X dada, solo habrá una Y.

De acuerdo a J. Hunt (2018), las funciones solo dependen de sus entradas y generarán una sola salida, teniendo las siguientes características :

1. **No tiene efectos secundarios:** Una función debe tener una transparencia referencial, es decir no debe de tener efectos secundarios ocultos, ya que su comportamiento no debe depender del estado del sistema que la vuelva impredecible es su resultado, esto es cuando se ejecute con los mismos datos de

entrada, siempre debe de regresar el mismo resultado, siendo funciones puras, las cuales tienen los siguientes atributos:

   a. La única salida observable es el valor de retorno.
   b. La única dependencia de salida son los argumentos.
   c. Los argumentos se determinan completamente antes de que se genere cualquier salida.
   d. La función solo hace algo en específico.

2. **No guardan estado:** No depende del estado del sistema o algún elemento del mismo, ya que las operaciones no están basadas en estados del sistema, lo único que depende de ello son los datos que se le proporcionan, haciendo que las funciones sean más fácil de entender, implementar, probar y depurar.
3. **Los datos son inmutables:** Una vez originado el dato, este no cambia su valor, ni su tipo, con esto se garantiza que las funciones no tengan efectos secundarios.
4. **Programación declarativa:** La programación se va realizando con sentencias que describen la solución de lo que se necesita que haga el programa.

De acuerdo a Trivodaliev, K. et al (2017), la influencia de la PF en el desarrollo de software ha incrementado al paso de los años, desde su introducción en 1960, su mención en los trabajos de investigación se ha incrementado mostrando evidencia de su alto impacto en el diseño, implementación, pruebas y mantenimiento en el desarrollo de software. Esto lo vemos reflejado en la Fig. 2.9, donde se muestra la frecuencia de publicación de papers por año.

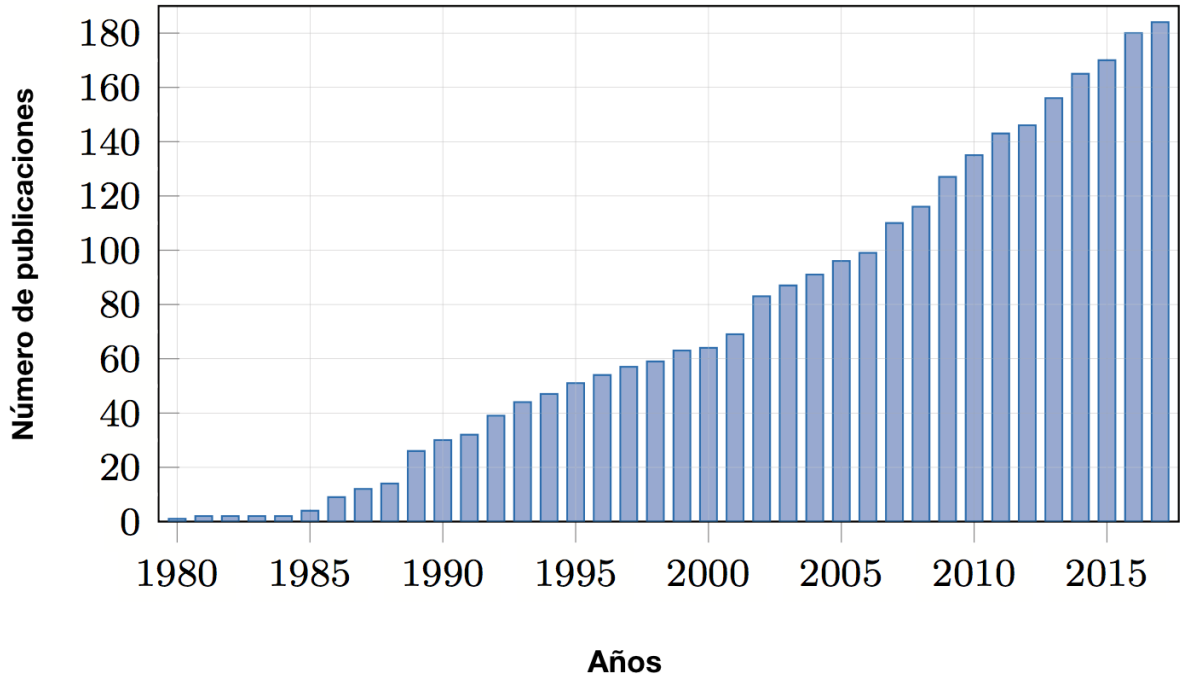

Figura 2.9.*Frecuencia acumulada de trabajos a lo largo de los años de publicación basada en la búsqueda bibliográfica.* Adoptado de "Trivodaliev, K., Stojkoska, B. R., Mihova, M., Jovanov, M., & Kalajdziski, S. (2017). Teaching Computer Programming : The Macedonian Case Study of Functional Programming, (April), 1282–1289.".

Ejemplos de lenguajes funcionales

Haskell

En 1936 el matemático y lógico norteamericano Alonzo Church propuso un enfoque diferente para el mismo problema que Turing intentaba resolver. Este enfoque, llamado cálculo lambda, dio como resultado el lenguaje LISP. Haskell deriva de LISP, teniendo la mayoría de las ventajas de este con el beneficio adicional de ser un lenguaje de programación puramente funcional y fuertemente tipado.

El trabajo en Haskell comenzó en septiembre de 1987 cuando un comité de investigadores se reunió después de la Conferencia de Programación Funcional y Arquitectura de Computadores (FPCA) en Portland, Oregon, para diseñar un lenguaje funcional único. En 2003 se publicó el Informe Haskell, que define una versión estable del lenguaje.

Las principales características de Haskell son:

- Las funciones no tienen efectos secundarios.
- No ejecuta funciones ni realiza ningún cálculo hasta que va a presentar algún resultado.
- Permite infinitas estructuras de datos.
- Es estrictamente tipificado: cada que se compila, el compilador conoce específicamente que tipo de dato existe en el código.
- Tiene inferencia de tipos: no se le tiene que indicar que tipos de datos estas en el código, ya que el sistema de tipos los identifica de manera inteligente.
- Utiliza conceptos de alto nivel: se escriben menos líneas de código.

Scala

Scala es un lenguaje desarrollado en Lausanne Suiza en el 2001 por Martin Odersky (anteriormente trabajó en Java genéricos y en el compilador de Java para Sun Microsystems) y su equipo de EPFL (*Ecole Polythenique Fererale* de Lausanne), se considera un lenguaje híbrido, ya que es un lenguaje funcional y orientado a objetos, las funciones y los valores son objetos (Hunt, J. 2018).

Es descrito como un lenguaje funcional ya que permite y promueve las técnicas de inmutabilidad y la programación sin efectos secundarios (Weston T. 2018).

De acuerdo a John Hunt (2018) Scala:

- Proporciona conceptos orientados a objetos que incluyen clases, objetos, herencia y abstracción.
- Incluye rasgos que representan datos y comportamientos que pueden mezclarse en clases y objetos.
- Permite construir nuevas funciones a partir de funciones existentes.
- Utiliza variables tipificadas estáticamente y constantes con inferencia de tipo utilizada siempre que sea posible para evitar repeticiones innecesarias.

- Tiene interoperabilidad, en su mayoría, con Java.

Elixir

Elixir en un lenguaje de programación funcional creado por José Valim en el 2011. Nace con la idea de escribir aplicaciones web altamente concurrentes así como el uso de la metaprogramación mediante el polimorfismo (Dadhich, S. et al 2017). Se ejecuta el la plataforma de Erlang (llamada la máquina virtual de Erlang o BEAM) y tiene características similares a Haskell (Cunningham, H. C. 2017).

3. Metodología

Para de desarrollar esta propuesta seguiremos una metodología inspirada por el análisis de patrones, “los patrones no son ‘inventados’ son ‘encontrados’ en modelos existentes” (Gamma, Helm, Johnson y Vlissides, 2005). Por ello el proceso de desarrollo de un patrón está compuesto de Identificación, Recolección y Codificación (Seruca & Loucopoulos, 2003). Por ella para este reporte técnico proponemos el siguiente proceso:

I. Identificación
   1. Selección del lenguaje funcional más popular.
   2. Selección de los 10 proyectos más populares.
   3. Identificación de las herramientas de IC usadas en los proyectos.

II. Recolección
   4. Análisis de herramientas encontradas en la inspección.

III. Codificación
   5. Propuesta de pipeline de IC base para el lenguaje de programación funcional.

La metodología que seguiremos está resumida en la siguiente gráfica:

Figura 3.1 ***.Metodología a seguir en el reporte técnico.*** Elaboración propia “Chaires K. 2019”.

El desarrollo de la metodología de este reporte está conformada de los siguientes puntos:

1. Selección del lenguaje funcional más popular.
2. Selección de los 10 proyectos más populares en el lenguaje funcional más popular.
3. Inspección de los archivos de configuración de IC , de los proyectos más populares.
4. Identificación de Herramientas de IC encontradas en la Inspección.
5. Propuesta de pipeline de IC.

4. Resultados

El desarrollo de la metodología de este reporte está conformada de los siguientes puntos:

1. Selección del lenguaje funcional más popular.

Basado en la revisión sistemática realizada por Abdullah Khanfor y Ye Yang (2017), la cual tiene como objetivo conocer el impacto práctico del uso del paradigma de programación funcional mediante la recopilación de los 10 principales lenguajes de programación que soportan este paradigma, los resultados obtenidos se muestran en la Fig 4.1, por un lado de acuerdo al ranking de lenguajes de programación “TIOBE Index” en sus resultados de agosto del 2017 el lenguaje de programación más utilizado es Python, en contraste con el estudio que llevaron a cabo en el cual muestra que Haskell es el lenguaje de programación más utilizado y/o estudiado (Khanfor, A. 2017).

Programming Language		
#	**TIOBE Index Rank**	**Literature Rank**
1	Python	Haskell
2	PHP	Java
3	JavaScript	Clojure
4	Swift	Erlang, Lisp, and OCaml
5	R	Merinda
6	Dart	Scala and SML
7	D	C++, F#, and Scheme
8	Scala	Bootstrap
9	Prolog	C#
10	Erlang	Clean

Figura 4.1. ***Lenguajes de programación funcional basados en el TIOBE INDEX RANK y la frecuencia de lenguajes en la literatura coleccionada.*** Adoptado de “ Khanfor, A. (2017). An Overview of Practical Impacts of Functional Programming. https://doi.org/10.1109/APSECW.2017.27”.

De acuerdo al reporte presentado en junio del 2017 por RedMonk Programming Language Rankings, el cual proporciona métricas de la clasificación de los lenguajes de programación, basado en la extracción combinada de la información de GitHub y Stack Overflow con el objetivo de proporcionar la correlación existente entre lenguajes que se discuten en Stack Overflow y el uso de los mismos en GitHub, obteniendo las futuras tendencias de adopción de los lenguajes encontrados en su estudio, en la Fig. 4.2, podemos observar los resultados obtenidos, (es importante mencionar que este estudio no es basado en lenguajes funcionales solamente) .

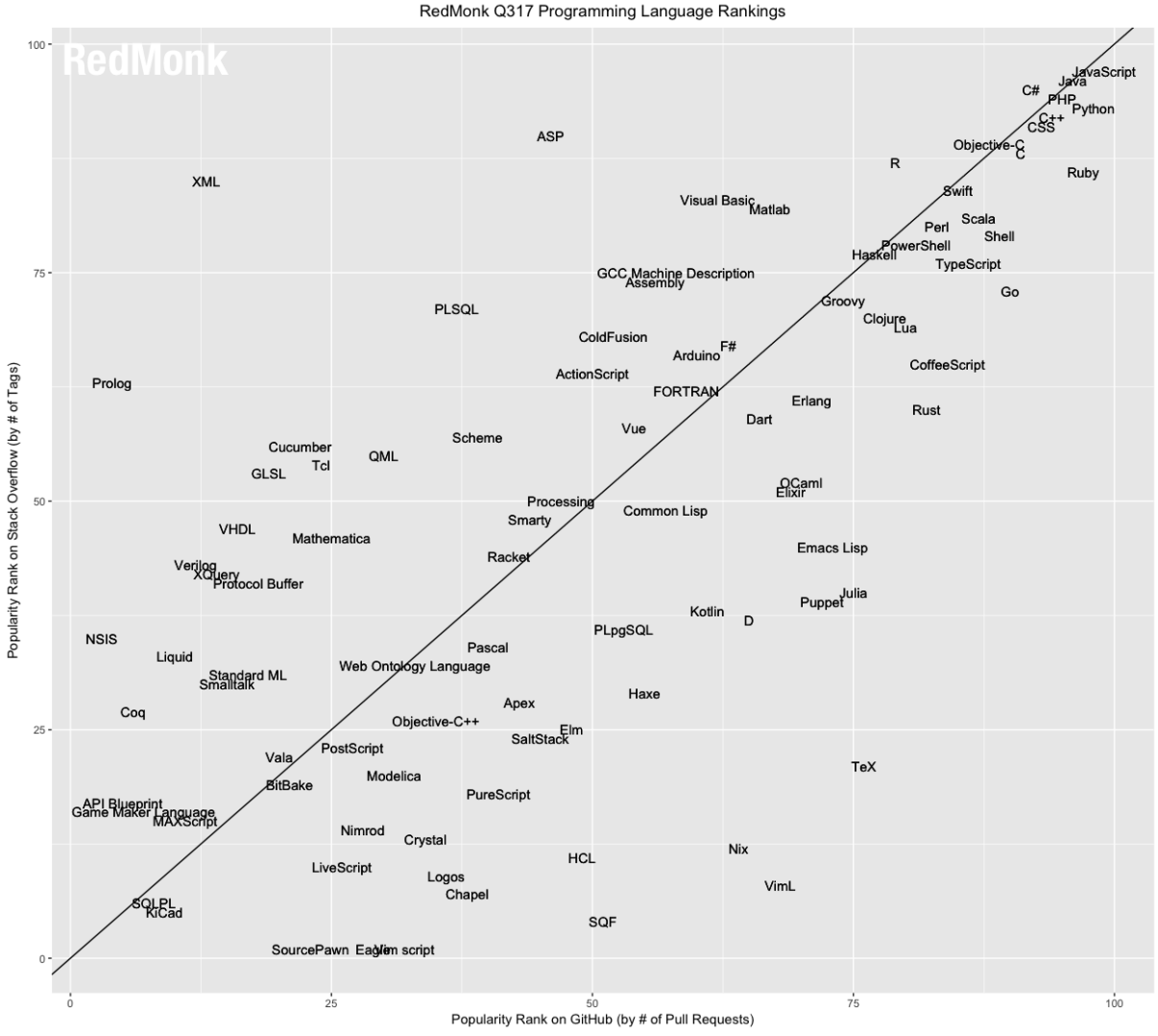

Figura 4.2. ***Ranking del lenguaje de programación RedMonk.*** Adoptado de "The RedMonk Programming Language Rankings: June 2017 – tecosystems. (n.d.). Retrieved from http://redmonk.com/sogrady/2017/06/08/language-rankings-6-17/".

Lista de resultados:

1 JavaScript

2 Java

3 Python

4 PHP

5 C #

6 C ++

7 CSS

8 Ruby

9 C

10 Objective-C

11 Swift

12 Shell

12 Scala

14 R

15 Go

15 Perl

17 TypeScript

18 PowerShell

19 Haskell

20 CoffeeScript

20 Lua

20 Matlab

De acuerdo a la lista de resultado, los lenguajes clasificados dentro de los primeros 20 que pertenecen al paradigma funcional son Scala y Haskell. Por lo tanto nuestro lenguaje de programación seleccionado es Haskell, ya que tanto en la literatura y en la práctica este lenguaje funcional es el más utilizado.

2. Selección de los 10 proyectos más populares.

Para los proyectos de desarrollo de software conformados por equipos distribuidos es ampliamente usado el modelo de desarrollo basado en ***pull request*** para integrar los cambios al código base del proyecto. Este modelo tiene la ventaja como lo es la centralización de la información y la automatización de procesos. Plataformas como GitHub integran funcionalidad para la generación de pull request, pruebas automáticas, revisión de código (Yu, Y. et al 2015).

Es así que los integradores de código a través de los ***pull request***, proporcionan el manejo de mecanismos que aseguran la calidad de software mediante las pruebas automáticas dando el respaldando a los servicios de integración continua (Yu, Y. et al 2015).

En esta fase aplicaremos la técnica de MSR, para extraer información de los registros de los eventos de pull request (Poncin W. et al 2011) producidos por los proyectos de Haskell, realizaremos dicha extracción de GitHub, ya que es uno de los repositorios de código abierto mayormente utilizado en todo el mundo (Borges, H. et al 2017). En la Fig. 4.3 mostramos un ejemplo de la estructura del archivo *githubarchive*, el cual contiene el archivo histórico de GitHub.


```
{
  "action": "opened",
  "number": 1,
  "pull_request": {
    "url": "https://api.github.com/repos/baxterthehacker/public-repo/pulls/1",
    "id": 34778301,
    "html_url": "https://github.com/baxterthehacker/public-repo/pull/1",
    "diff_url": "https://github.com/baxterthehacker/public-repo/pull/1.diff",
    "patch_url": "https://github.com/baxterthehacker/public-repo/pull/1.patch",
    "issue_url": "https://api.github.com/repos/baxterthehacker/public-repo/issues/1",
    "number": 1,
    "state": "open",
    "locked": false,
    "title": "Update the README with new information",
    "user": {
```


Figura 4.3. ***Review Requests***. Adoptado de "Review Requests | GitHub Developer Guide. (n.d.). Retrieved from https://developer.github.com/v3/pulls/review_requests//".

Para nuestro caso definiremos la instancia de los eventos de pull request ocurridos en los proyectos desarrollados con Haskell en el periodo del 1 de enero del 2017 al 31 de mayo del 2018, fecha en la que se realizó el análisis, seleccionando el nombre del repositorio y el total de eventos de pull request con los que cuenta.

Esto lo realizamos mediante Google BigQuery el cual es un almacén de datos empresariales que permitir consultas de SQL súper rápidas utilizando el poder de procesamiento de la infraestructura de Google. Podemos utilizar la interfaz de usuario web que nos proporciona acceso a la base de datos del archivo histórico de GitHub (https://cloud.google.com/bigquery/what-is-bigquery).

En la Fig. 4.4 se visualizan los pasos ejecutados en la interfaz de usuario web mencionada anteriormente; primero del evento de pull request seleccionamos el nombre del lenguaje de programación (*events_repo_language*), el nombre del repositorio (events.repo.name) y el total de eventos de pull request (events.type = '*PullRequestEvent*') donde la fecha del evento este entre 01 de enero del 2017 al 31 de mayo del 2018 y el lenguaje de programación sea Haskell (pull_request.base.repo.language).

```
-- Top ten de proyectos de Haskell más populares por Pull requests
-- Principales proyectos de programación activos por número de pull requests en el marco de tiempo especificado
-- Fuente: githubarchive public data set via Google BigQuery http://githubarchive.org/

SELECT
  JSON_EXTRACT_SCALAR(events.payload, '$.pull_request.base.repo.language') AS events_repo_language,
  events.repo.name AS events_repo_name,
  COUNT(CASE WHEN (events.type = 'PullRequestEvent') THEN 1 ELSE NULL END) AS events_count_pull_request
FROM (
  SELECT
    *
  FROM TABLE_DATE_RANGE([githubarchive:day.],TIMESTAMP('2017-01-01'),TIMESTAMP('2018-05-31'))) AS events
WHERE
  JSON_EXTRACT_SCALAR(events.payload, '$.action') = 'opened'
  AND JSON_EXTRACT_SCALAR(events.payload, '$.pull_request.base.repo.language') ='Haskell'
GROUP BY
  1,2
ORDER BY
  3 DESC
LIMIT
10
```

Figura 4.4. ***Consulta para obtener los 10 principales proyectos de Haskell.*** Elaboración propia "Chaires K. 2019".

En la tabla 4.1 presentamos el resultado de la consulta realizada, la columna *"events_repo_name"* es la que contiene el nombre del proyecto en GitHub, con esta información, procederemos a consultar y analizar los archivos contenidos en cada repositorio.

events_repo_language	events_repo_name	events_count_pull_request
Haskell	input-output-hk/cardano-sl	2182
Haskell	haskell/cabal	526
Haskell	databrary/databrary	441
Haskell	purescript/purescript	427
Haskell	idris-lang/Idris-dev	412
Haskell	commercialhaskell/stack	344
Haskell	jgm/pandoc	324
Haskell	futurice/haskell-mega-repo	310
Haskell	wireapp/wire-server	253
Haskell	TorXakis/TorXakis	216

Tabla 4.1. ***Lista de los 10 proyectos de Haskell con mayor número de PullRequest en GitHub.*** Elaboración propia "Chaires K. 2019".

3. Identificación de las herramientas de IC usadas en los proyectos.

Para cada uno de los repositorios, se realizó el análisis manual de los archivos, con el objetivo de obtener y clasificar las herramientas que de acuerdo a su configuración e implementación para llevar a cabo cada una de las fases de la IC, en la Fig. 4.5 se muestra un ejemplo del contenido de estos archivos.

```
146  tests:
147    tests:
148      main: Main.hs
149      source-dirs: tests
150      ghc-options: -Wall
151      dependencies:
152        - purescript
153        - tasty
154        - tasty-hspec
155        - hspec
156        - hspec-discover
157        - HUnit
158      default-extensions:
159        - NoImplicitPrelude
160
```

Figura 4.5. ***Fragmento de un archivo de configuración.***

4. Análisis de herramientas encontradas en la inspección.

Una vez analizados cada uno de los archivos de configuración de los proyectos, procedimos a realizar la clasificación en cada una de las fase del pipeline propuesto por Shahin, M. et al (2017). La Fig. 4.6 nos presenta estas fases, así como las herramientas utilizadas en cada fase por los lenguajes de programación procedurales.

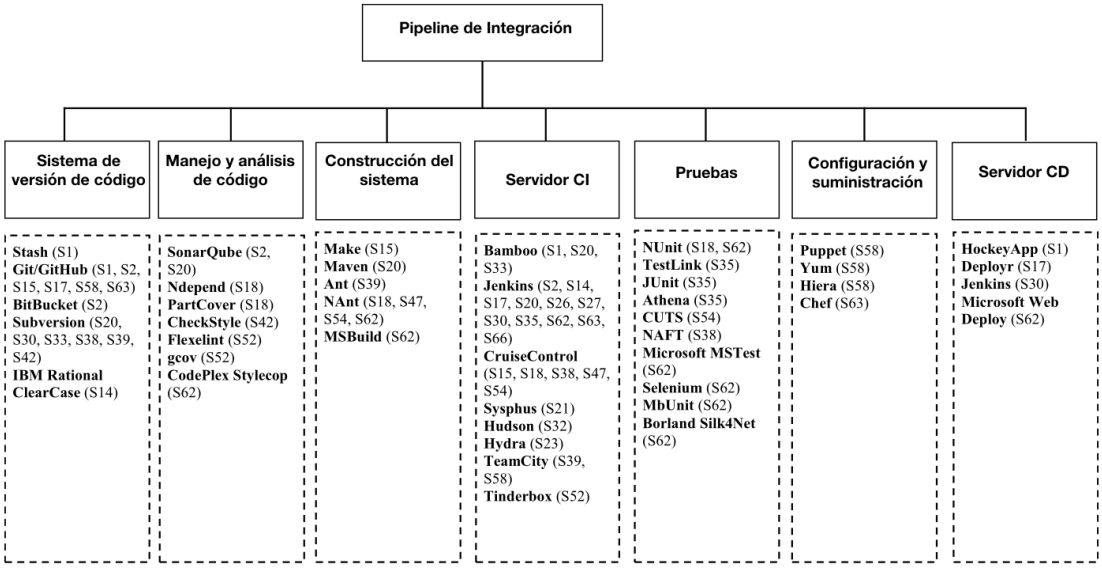

Figura. 4.6.*Vista general de las herramientas utilizadas para formar el pipeline de integración.* Adoptado de "Shahin, M., Babar, M. A. L. I., & Zhu, L. (2017). Continuous Integration , Delivery and Deployment : A Systematic Review on Approaches , Tools , Challenges and Practices, (Ci), 3909–3943./".

En la tabla 4.2, se muestran el resultado de la clasificación de las herramientas que encontramos para cada uno de los proyectos de Haskell analizados.

Proyecto	Sistema de versión de código	Manejo y análisis de código	Construcción del sistema	Servidor CI	Testing	Configuración y suministración	Servidor CD
input-output-hk /cardano-sl	Git/GitHub	HLint stylish-haskell shellcheck walletIntegration swaggerSchemaValidation yamlValidation	Stack	AppVeyor	QuickCheck Hspec crypto/test binary/test util/test infra/test chain/test	nix	Hydra
haskell /cabal			Stack		QuickCheck	nix	
databrary /databrary	Git/GitHub	HLint	Cabal		QuickCheck Hspec hspec-expectations tasty tasty-discover tasty-expected-failure tasty-hedgehog tasty-hunit tasty-quickcheck	nix	
purescript /purescript	Git/GitHub	HLint	Stack	Travis CI	tasty tasty-hspec Hspec hspec-discover HUnit		
idris-lang /Idris-dev	Git/GitHub	HLint	Make Cabal Stack	Travis CI AppVeyor	tasty	nix	
commercialhaskell /stack	Git/GitHub	HLint	Stack Cabal	Travis CI AppVeyor	Hspec		
jgm /pandoc	Git/GitHub	HLint	Cabal	Travis CI AppVeyor	tasty Tasty.HUnit	nix	
futurice /haskell-mega-repo	Git/GitHub	doctest	Cabal	Travis CI	HUnit QuickCheck Hspec tasty		
wireapp /wire-server	Git/GitHub		Make		QuickCheck tasty-cannon		
TorXakis /TorXakis	Git/GitHub		Stack	AppVeyor	HUnit		

Tabla 4.2. ***Clasificación de herramientas en cada una de las fases del pipeline***. Elaboración propia “Chaires K. 2019”.

Como podemos observar, no todos los proyectos cumplen con el pipeline completo, solo uno que es el de “cardano-sl”, en la Fig. 4.7 mostramos el número de fases que cumple cada uno de los proyectos.

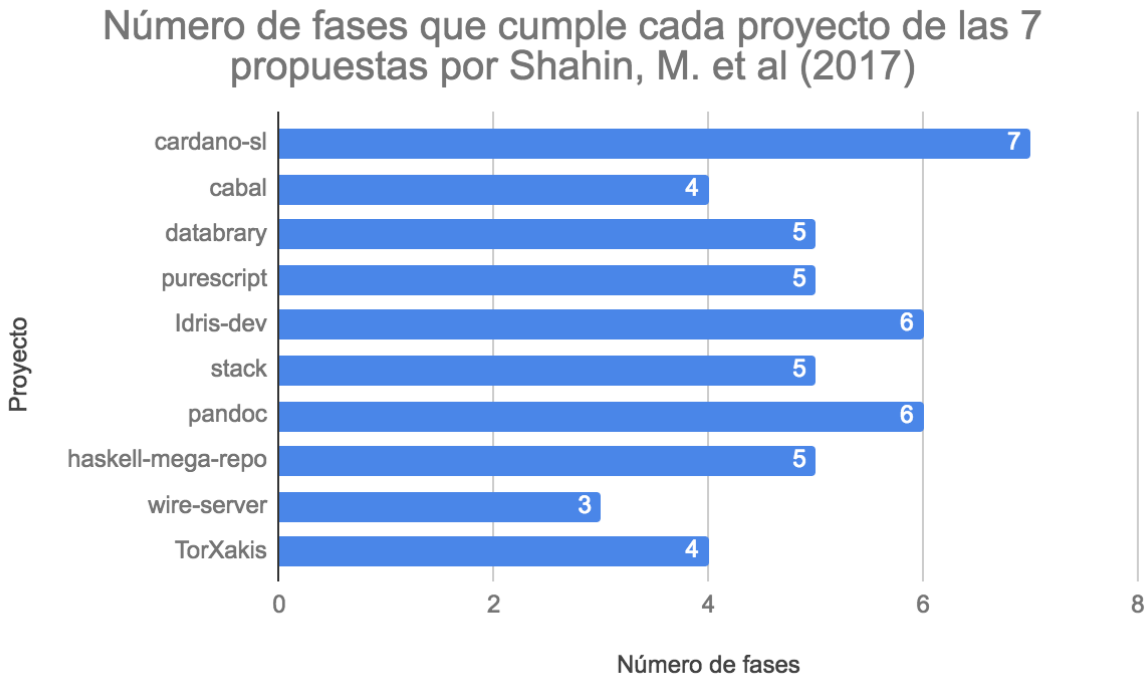

Figura 4.7. ***Número de fases que cumple cada proyecto.*** Elaboración propia “Chaires K. 2019”.

5. Propuesta de pipeline de IC base para el lenguaje de programación funcional.

En la Fig 4.8 podemos observar el pipeline de IC que de acuerdo a la práctica y la literatura, para cada fase son las herramientas de IC que los proyectos de Haskell pueden implementar para así asegurar la calidad del software que esten desarrollando.

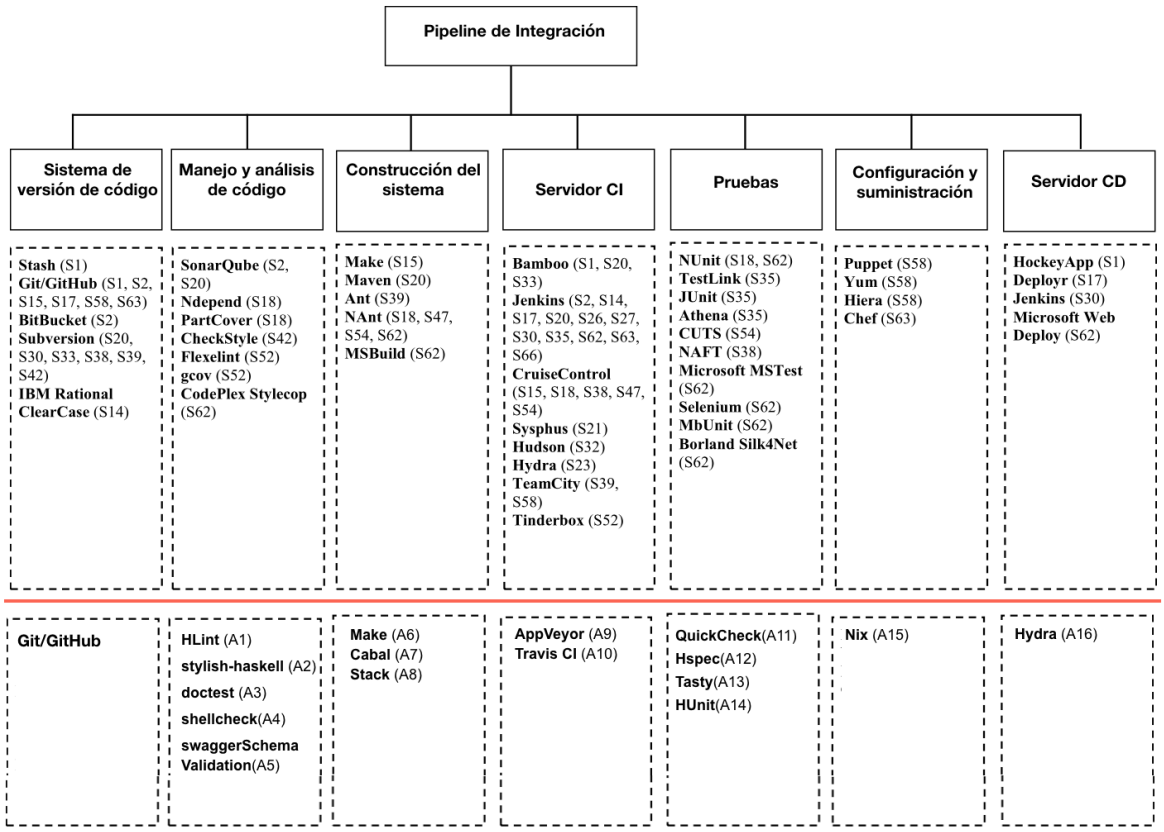

Figura 4.8.Propuesta de pipeline de IC para proyectos funcionales. Elaboración propia “Chaires K. 2019”.

5. Conclusiones, discusión y trabajo futuro

A continuación se presenta las conclusiones de este reporte:

Respecto al Objetivo general, se obtuvo un pipeline de integración continua para lenguajes funcionales.

De los 10 proyectos con mayor actividad en la plataforma más popular de repositorios de código, solo 1 utiliza herramientas para las 7 fases del pipeline (Cardano SL).

Respecto a las preguntas de investigación:

Q1. Se comprobó que la integración continua es utilizada en los 10 proyectos con más actividad de manera parcial.

Q2. Dentro de cada una de las fases de integración continua existen una o más herramientas utilizada en los 10 proyectos más populares. Estas están plasmadas en el pipeline de integración continua resultado de este trabajo. Por tanto, sí fueron detectadas y mapeadas contra cada una de las fases del pipeline.

Q3. Con base a la investigación realizada, si existe un pipeline de integración continua para lenguajes procedurales, el cual se tomó como base para poder integrar el pipeline de lenguajes funcionales.

Q4. En base al estudio realizado, si se requiere contar con un pipeline base de IC en lenguajes funcionales, ya que al implementar las herramientas base en cualquier proyecto, asegura la calidad del mismo.

Lecciones aprendidas

En definitiva en necesario contar con herramientas sustentadas en la literatura y en la práctica en la industria, pudimos observar que el llevar a cabo una investigación con una metodología ayuda a obtener resultados sustentados, en próximas trabajos cambiariamos la forma de implentar el análisis, auque llevaría más tiempo desarrollar el proceso automatizado sería de gran utilizadad para poder llevar a cabo métodos de correlación entre los diferentes repositorios de código existentes.

Trabajo futuro

- Realizar la automatización del proceso de inspección de los archivos de configuración, para obtener las herramientas de pruebas utilizadas.
- Hacer un curso presencial para la implementación de las herramientas propuestas.
- Realizar un comprativo he implementación de este trabajo en un proyecto en producción.

6. Anexos

A1. HLint

Es una herramienta para analizar y sugerir mejoras al código escrito en Haskell. Las sugerencias pueden ser de estilo, paréntesis innecesarios o funciones alternativas. Usar esta herramienta permite desarrollar códigos más cortos y performantes (Putrady 2018).

A2. Stylish Haskell

Es una herramienta de Haskell que lee el código y produce una nueva versión siguiendo una serie de opciones configurables (Mena 2014).

Características:

- Alinea y ordena las declaraciones de importación
- Grupos y ajustes {- # IDIOMA # -} pragmas, pueden eliminar algunos programas redundantes.
- Elimina los espacios finales en blanco.
- Alinea ramas en caso y campos en registros.
- Convierte los finales de línea (personalizables).
- Reemplaza las pestañas por cuatro espacios.
- Reemplaza algunas secuencias ASCII por sus equivalentes de Unicode.

(Jaspervdj 2019)

A3. DocTest

Es una librería de verificación de pruebas dinámicas, basado en la librería doctest de Python (Hengel, S. (n.d.)), ejecuta pruebas que estén en los comentarios del código, con formato Haddock la cual es una librería de Haskell para realizar documentación (doctest: Test interactive Haskell examples. (n.d.)).

En la Fig. 7.1 se muestra un ejemplo de un módulo que implementa este tipo de pruebas:

```haskell
module Fib where

-- | Compute Fibonacci numbers
--
-- Examples:
--
-- >>> fib 10
-- 55
--
-- >>> fib 5
-- 5
fib :: Int -> Int
fib 0 = 0
fib 1 = 1
fib n = fib (n - 1) + fib (n - 2)
```

Figura 7.1.*Ejemplo de implementación* . Adoptado de “GitHub - sol/doctest: An implementation of Python’s doctest for Haskell. (n.d.). Retrieved from https://github.com/sol/doctest#readme”.

A4. Shellcheck

Es un linter para bash y shell scripts de análisis estático de código, el cual analiza código defectuoso realizando la lectura linea por linea, comprobando solo la codificación no la ejecución (Ho et al 2018).

A5. SwaggerSchemaValidation

Son funciones diseñadas para usarse en suites de pruebas útiles para asegurar que la representación de los esquemas JSON sea válida con respecto a la especificación de Swagger (Data.Swagger.Schema.Validation. (n.d.)).

A6. Make

Librería para realizar la construcción del sistema, definiendo reglas de construcción, proveyendo de los estándares disponibles para la construcción del sistema.

A7. Cabal

Cabal es un sistema para construir y empaquetar librerías y programas de Haskell. Esta herramienta define una interfaz común para que los autores y distribuidores de paquetes pueden construir fácilmente sus aplicaciones de manera portátil (https://www.haskell.org/cabal/).

A8. Stack

Es un constructor de programación que soporta las tareas de depuración, útil para identificar la causa del defecto entre millones de líneas de código (Schröter et al 2010).

A9. AppVeyor

Es una plataforma de CI/CD, se pueden alojar repositorios de GitHub e integrarlos con AppVeyor para realizar la construcción automatizada desde un commit (Edition, S. 2016).

Los repositorios que soporta son: GitHub, GitHub Enterprise, Bitbucket, GitLab, VSTS, Kiln o repositorios propios.

- El archivo de configuración es atreves de YAML o UI.
- Para cada construcción, se construyen ambientes aislados y limpios.
- Implementación incorporada y servidor NuGet.
- Branch y PR se construyen para soportar su flujo de trabajo de desarrollo.
- Apoyo personal y activo para la comunidad.

(Continuous Integration and Deployment service for Windows and Linux | AppVeyor. (n.d.))

A10. Travis CI

Es un servicio de integración continua distribuido para alojar, construir y probar la funcionalidad del código. Esto a través del repositorio de código como lo GitHub, soporta la compilación en diversos entornos. La implementación puede ser en AWS, Azure, Heroku y npm. Ejecuta construcciones simultáneas , también puede realizar la cancelación

automática de la compilación (Configuring a Continuous Build with Travis CI. (2017)), en la Fig. 7.2 se muestra su proceso.

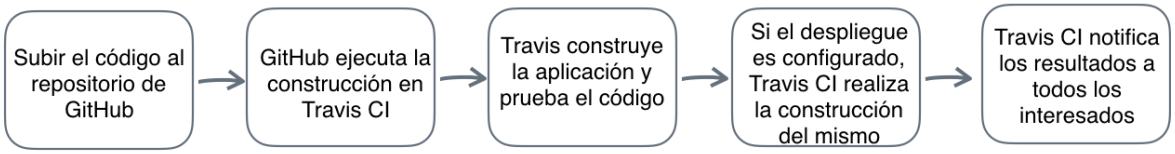

Figura 7.2 .*Proceso de Travis CI.* Adoptado de "Configuring a Continuous Build with Travis CI. (2017)".

A11. QuickCheck

Ayuda al programador de Haskell a probar automáticamente las propiedades de las funciones con entradas aleatorias, así como definir generadores con datos de pruebas personalizados.

A12. Hspec

Hspec es un framework de pruebas para Haskell, algunas de las características de esta herramienta son:

- Definición de pruebas usando un lenguaje de dominio específico (DSL).
- Integración con with *QuickCheck*, *SmallCheck,* y *HUnit*.
- Ejecución de pruebas en paralelo.
- Descubrimiento automático de archivos de prueba.

(Hspec: A Testing Framework for Haskell. (n.d.))

Hspec usa un estilo más textual para describir pruebas. el objetivo es acercarse más al lenguaje utilizado en la fase de especificación y así facilitar la verificación de una implementación (Mena 2014).

A12.1. Hspec-discover

Es una característica del framework Hspec, con la cual las pruebas son automáticamente añadidas en la ejecución de otras pruebas. Esto reduce la cantidad de código repetitivo. Para habilitar esa función, es necesario cambiar el contenido de test/Spec.hs y agregar la siguiente línea:

{-# OPTIONS_GHC -F -pgmF hspec-discover #-}

(Putrady 2018)

A13. Tasty

Es un moderno framework de pruebas para Haskell, que permite combinar pruebas unitarias, pruebas doradas (pruebas de entrada/salida que son guardadas y comparadas en archivos), pruebas de propiedades QuickCheck/SmallCheck, entre otros tipos de pruebas en una única suite de pruebas.

Características:

- Ejecuta las pruebas en paralelo reportando los resultados en orden determinístico.
- Filtra las pruebas que se ejecutarán usando patrones específicos en la línea de comandos.
- Visualización jerárquica, coloreada de resultados de pruebas.
- Reporte de estadísticas de pruebas.
- Genera y libera recursos (sockets, archivos temporales, etc.) que pueden ser compartidos entre varias pruebas.
- Extensibilidad: tiene la funcionalidad de agregar otros tipos de pruebas por encima de los proporcionados.

(Tasty: Modern and extensible testing framework. (n.d.))

- tasty-hunit: para pruebas unitarias (basadas en HUnit).

- tasty-golden: para las pruebas superiores, que son pruebas unitarias cuyos resultados se guardan en archivos.
- tasty-smallcheck: pruebas exhaustivas basadas en propiedades (basadas en smallcheck).
- tasty-quickcheck: para pruebas aleatorias basadas en propiedades (basadas en QuickCheck).
- tasty-hedgehog: para pruebas aleatorias basadas en propiedades (basadas en Hedgehog).
- tasty-hspec: para pruebas Hspec.
- tasty-leancheck: para pruebas basadas en propiedades enumerativas (basadas en LeanCheck).
- tasty-program: ejecuta un programa externo y comprueba si finaliza correctamente.

(Tasty: Modern and extensible testing framework. (n.d.))

A14. HUnit

HUnit es una framework de pruebas para Haskell, inspirado en la herramienta JUnit de Java. Una metodología centrada en pruebas de desarrollo de software es más efectiva cuando las pruebas son fáciles de crear, modificar y ejecutar.

Con HUnit al igual que con JUnit, es posible crear pruebas, nombrarlas, agruparlas y ejecutarlas. La especificación de pruebas en HUnit es aún más concisa y flexible que en JUnit, debido a la naturaleza del lenguaje Haskell (HUnit: A unit testing framework for Haskell. (n.d.)).

A15. Nix

Nix es un gestor de paquetes que tiene un enfoque en la fiabilidad y reproducibilidad, se integra co

n cabal para la administración de dependencias durante el desarrollo de paquetes.

(Nix integration - The Haskell Tool Stack. (n.d.))

A16. Hydra

Hydra es una herramienta para la integración continua de pruebas así como el despliegue de nuevas versiones de código, realiza la construcción del código y sus dependencias. Hydra promueve la calidad en los procesos de desarrollo de software, ya que provee un sistema de automatización continua así como la verificación periódica del código del proyecto, la construcción y ejecución de pruebas. Proporciona reportes para los desarrolladores.

(Hydra - NixOS Wiki. (n.d.))

7. Referencias

Shahin, M., Babar, M. A. L. I., & Zhu, L. (2017). Continuous Integration , Delivery and Deployment : A Systematic Review on Approaches , Tools , Challenges and Practices, (Ci), 3909–3943.

Ståhl, D., & Bosch, J. (2017). Cinders: The continuous integration and delivery architecture framework. Information and Software Technology, 83, 76–93. https://doi.org/10.1016/j.infsof.2016.11.006

Hilton, M., Tunnell, T., Huang, K., Marinov, D., & Dig, D. (2016). Usage, costs, and benefits of continuous integration in open-source projects. Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering - ASE 2016, 426–437. https://doi.org/10.1145/2970276.2970358

Bollati, V. A., Gaona, G., Pletsch, L. C., Gonnet, S., & Leone, H. (2017). The state of agile development adoption in Argentine software companies. 2017 43rd Latin American Computer Conference, CLEI 2017, 2017–January, 1–10. https://doi.org/10.1109/CLEI.2017.8226394

Michelsen, J. (2014). A Pragmatic Guide to Getting Started with DevOps. CA Technologies, 26. https://doi.org/10.1017/S0031182000064192

L.Bass, I.Weber, and L.Zhu, DevOps: A Software Architect's Perspective. Reading, MA, USA: Addison-Wesley, 2015.

Static Code Analysis - OWASP. (n.d.). Retrieved from https://www.owasp.org/index.php/Static_Code_Analysis

What is Continuous Integration in Agile methodology? (n.d.). Retrieved from http://tryqa.com/what-is-continuous-integration-in-agile-methodology/

Felbinger, H., Wotawa, F., & Nica, M. (2018). Adapting unit tests by generating combinatorial test data. Proceedings - 2018 IEEE 11th International Conference on Software Testing, Verification and Validation Workshops, ICSTW 2018, 352–355. https://doi.org/10.1109/ICSTW.2018.00072

Eddy, B. P., Wilde, N., Cooper, N. A., Mishra, B., Gamboa, V. S., Shah, K. M., … Shields, N. A. (2017). A Pilot Study on Introducing Continuous Integration and Delivery into Undergraduate Software Engineering Courses. Proceedings - 30th IEEE Conference on Software Engineering Education and Training, CSEE and T 2017, 2017–January, 47–56. https://doi.org/10.1109/CSEET.2017.18

Marijan, D., Liaaen, M., & Sen, S. (2018). DevOps Improvements for Reduced Cycle Times with Integrated Test Optimizations for Continuous Integration. 2018 IEEE 42nd Annual Computer Software and Applications Conference (COMPSAC), 22–27. https://doi.org/10.1109/COMPSAC.2018.00012

Lu, H., Lin, P., Huang, P., & Yuan, A. (2017). Deployment and Evaluation of a Continues Integration Process in Agile Development, 8(4), 204–209. https://doi.org/10.12720/jait.8.4.204-209

O. Lavriv, B. Buhyl, M. Klymash and G. Grynkevych, "Services continuous integration based on modern free infrastructure," 2017 2nd International Conference on Advanced Information and Communication Technologies (AICT), Lviv, 2017, pp. 150-153.

Düllmann, T. F., Paule, C., & Van Hoorn, A. (2018). Exploiting devops practices for dependable and secure continuous delivery pipelines. Proceedings - International Conference on Software Engineering, Part F1378, 27–30. https://doi.org/10.1145/3194760.3194763

Borges, H., Hora, A., & Valente, M. T. (2017). Understanding the factors that impact the popularity of GitHub repositories. Proceedings - 2016 IEEE International Conference on Software Maintenance and Evolution, ICSME 2016, (Dcc), 334–344. https://doi.org/10.1109/ICSME.2016.31

Hu, Y., Zhang, J., Bai, X., Yu, S., & Yang, Z. (2016). Influence analysis of Github repositories. SpringerPlus, 5(1). https://doi.org/10.1186/s40064-016-2897-7

Cosentino, V., & Izquierdo, C. (2016). Findings from GitHub : Methods , Datasets and Limitations, 137–141. https://doi.org/10.1145/2901739.2901776

Siddiqui, T., & Ahmad, A. (2018). Data mining tools and techniques for mining software repositories: A systematic review. Advances in Intelligent Systems and Computing, 654, 717–726. https://doi.org/10.1007/978-981-10-6620-7_70

Aravinth A., Machiraju S. (2018) Functional Programming in Simple Terms. In: Beginning Functional JavaScript. Apress, Berkeley, CA

Hunt, J. (2018). A Beginner's Guide to Scala, Object Orientation and Functional Programming. https://doi.org/10.1007/978-3-319-06776-6

Trivodaliev, K., Stojkoska, B. R., Mihova, M., Jovanov, M., & Kalajdziski, S. (2017). Teaching Computer Programming : The Macedonian Case Study of Functional Programming, (April), 1282–1289.

Weston T. (2018) The Scala Language. In: Scala for Java Developers. Apress, Berkeley, CA

Dadhich, S., Dadhich, V., & Kumar, A. (2017). Elixir : A Functional Programming, 2(3), 698–701.

Cunningham, H. C. (2017). CSci 450 : Org . of Programming Languages Evaluation and Efficiency, (September).

Poncin, W., Serebrenik, A., & Brand, M. Van Den. (2011). Process mining software repositories. 2011 15th European Conference on Software Maintenance and Reengineering, 5–14. https://doi.org/10.1109/CSMR.2011.5

(n.d.). Retrieved from https://www.haskell.org/cabal/

HUnit: A unit testing framework for Haskell. (n.d.). Retrieved from http://hackage.haskell.org/package/HUnit

Hspec: A Testing Framework for Haskell. (n.d.). Retrieved from http://hspec.github.io/

Jaspervdj. (2019, January 02). Jaspervdj/stylish-haskell. Retrieved from https://github.com/jaspervdj/stylish-haskell

Mena, A. S. (2014). Documenting, Testing, and Verifying. Beginning Haskell, 355-371. doi:10.1007/978-1-4302-6251-0_15

Putrady, E. (2018). Testing. Practical Web Development with Haskell, 219-260. doi:10.1007/978-1-4842-3739-7_11

Tasty: Modern and extensible testing framework. (n.d.). Retrieved from http://hackage.haskell.org/package/tasty

Khanfor, A. (2017). An Overview of Practical Impacts of Functional Programming. https://doi.org/10.1109/APSECW.2017.27

Review Requests | GitHub Developer Guide. (n.d.). Retrieved from https://developer.github.com/v3/pulls/review_requests

What is BigQuery? | BigQuery | Google Cloud. (n.d.). Retrieved from https://cloud.google.com/bigquery/what-is-bigquery

Yu, Y., Wang, H., Filkov, V., Devanbu, P., & Vasilescu, B. (2015). Wait for It: Determinants of pull request evaluation latency on GitHub. IEEE International Working Conference on Mining Software Repositories, 2015–August, 367–371. https://doi.org/10.1109/MSR.2015.42

Sajedi Badashian, A., Esteki, A., Gholipour, A., Hindle, A., & Stroulia, E. (2014). Involvement, Contribution and Influence in Github and StackOverflow. Cascon, 19–33. https://doi.org/10.1143/JJAP.26.1978

Hengel, S. (n.d.). Behavior-Driven Development in Haskell, 1–5.

doctest: Test interactive Haskell examples. (n.d.). Retrieved from http://hackage.haskell.org/package/doctest-0.4.2

Ho, C. K., & Booi, L. L. (2018). Descriptive Review for Software Testing Algorithms, (December), 0–24. https://doi.org/10.13140/RG.2.2.11325.10724

Data.Swagger.Schema.Validation. (n.d.). Retrieved from http://hackage.haskell.org/package/swagger2-2.3.1/docs/Data-Swagger-Schema-Validation.html#g:2

Schröter, A., Bettenburg, N., & Premraj, R. (2010). Do stack traces help developers fix bugs? *Proceedings - International Conference on Software Engineering*, 118–121. https://doi.org/10.1109/MSR.2010.5463280

Edition, S. (2016). Pro PowerShell Desired State Configuration, 453–464. https://doi.org/10.1007/978-1-4842-3483-9

Continuous Integration and Deployment service for Windows and Linux | AppVeyor. (n.d.). Retrieved from https://www.appveyor.com/

Configuring a Continuous Build with Travis CI. (2017).

Nix integration - The Haskell Tool Stack. (n.d.). Retrieved from https://docs.haskellstack.org/en/stable/nix_integration/

Hydra - NixOS Wiki. (n.d.). Retrieved from https://nixos.wiki/wiki/Hydra

GitHub - sol/doctest: An implementation of Python's doctest for Haskell. (n.d.). Retrieved from https://github.com/sol/doctest#readme

Chaires, K.(2019, 1). Metodología a seguir en el reporte técnico.

Humble, J., & Kim, G. (2018). Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.

Fraser, S., Gamma, E., Helm, R., & Johnson, R. (2006). Design Patterns: Beginnings and Futures. Companion to the 21st ACM SIGPLAN Symposium on Object-Oriented Programming Systems, Languages, and Applications, 934. https://doi.org/10.1145/1176617.1176748

Seruca, I., & Loucopoulos, P. (2003). Towards a systematic approach to the capture of patterns within a business domain. Journal of Systems and Software, 67(1), 1–18. https://doi.org/10.1016/S0164-1212(02)00083-3

CIMAT

CENTRO DE INVESTIGACIÓN EN MATEMÁTICAS, A.C.

BIBLIOTECA

AUTORIZACION
PUBLICACION EN FORMATO ELECTRONICO DE TESIS

El que suscribe	Karina Daniela Chaires López
Autor(s) de la tesis:	"Orquestación de herramientas de integración continua en lenguajes funcionales"
Institución y Lugar:	CIMAT Zacatecas, Zacatecas
Grado Académico:	Licenciatura ( ) Maestría (x) Doctorado ( ) Otro ( ) ----------
Año de presentación:	2019
Área de Especialidad:	Ingeniería de Software
Director(es) de Tesis:	Maestro Alejandro García Fernández
Correo electrónico:	karina.chaires@cimat.mx
Domicilio:	And. Faro H-11 Rincón Colonial Guadalupe Zacatecas
Palabra(s) Clave(s):	Integración continua, despliegue continuo, orquestación, pipeline, lenguajes funcionales, Mining Software Repositories, GitHub, Haskell
ORCID:	0000-0002-6430-1254

Por medio del presente documento autorizo en forma gratuita a que la Tesis arriba citada sea divulgada y reproducida para publicarla mediante almacenamiento electrónico que permita acceso al público a leerla y conocerla visualmente, así como a comunicarla públicamente en la Página WEB del Repositorio Institucional y Repositorio Nacional.

La vigencia de la presente autorización es por tiempo indefinido a partir de la firma de presente instrumento, quedando en el entendido de que si por alguna razón el alumno desea revocar la autorización tendrá que hacerlo por escrito con acuse de recibo de parte de alguna autoridad del CIMAT

Atentamente

Karina Daniela Chaires López

Nombre y/o firma del tesísta

CALLE JALISCO S/N MINERAL DE VALENCIANA APDO. POSTAL 402
C.P. 36240 GUANAJUATO, GTO., MÉXICO
TELÉFONOS (473) 732 7155, (473) 735 0800 EXT. 49609 FAX. (473) 732 5749
E-mail: biblioteca@cimat.mx