FICHA RESUMEN
TRABAJO DE GRADO DE MAESTRÍA

TITULO: “Creación Automática de Pipelines Para CI/CD Usando Técnicas de Reutilización y Variabilidad”

1. ÉNFASIS: Ingeniería de Software
2. TIPO DE PROYECTO: Aplicado
3. ÁREA DE TRABAJO: CI/CD y Líneas de Productos de Software
4. ESTUDIANTE (S): Robinson Cruz Delgado
5. CORREO ELECTRÓNICO: robcruzd@javerianacali.edu.co
6. DIRECCIÓN Y TELÉFONO: Cra 19F # 27 – 33 Sur Bogotá D.C. cel. 3135818403
7. DIRECTOR: Jaime Alberto Chavarriaga Lozano
8. VINCULACIÓN DEL DIRECTOR (en la universidad): Cátedra
9. CORREO ELECTRÓNICO DEL DIRECTOR: jaime.chavarriaga@javerianacali.edu.co
10. CO-DIRECTOR(ES) (Si aplica): Luisa Fernanda Rincón
11. GRUPO O EMPRESA QUE LO AVALA (Si aplica):
12. OTROS GRUPOS O EMPRESAS:
13. PALABRAS CLAVE (al menos 5): Integración continua, despliegue continuo, pipeline, reutilización de software, variabilidad de software.
14. ODS QUE APLICA EL PROYECTO (Agenda 2030): Objetivo 9, Industria, Innovación e Infraestructura.
15. FECHA DE INICIO (Desarrollo del proyecto): 1/08/2022
16. RESUMEN (máximo 400 palabras): La integración y despliegue continuo, junto con los pipelines como código, han generado una revolución en la industria del software permitiendo publicar cambios desde el código en menos de una hora en producción. Sin embargo, los pipelines como código agregan los mismos problemas del código en general, como malas practicas de desarrollo y configuración, vulnerabilidades, desconfianza o sobrecostos durante su ejecución. Aunque, se han creado herramientas para generar pipelines e intentar solventar estos problemas, algunos se enfocan en una o dos herramientas de CI/CD y no brindan variedad y en otros casos solo se llega el pipeline hasta la fase de pruebas o entrega continua, pero no hasta el despliegue.

    Por ello, se creó una herramienta que aproveche las cualidades de la variabilidad y el reuso, para generar pipelines de CI/CD mitigando los problemas ya mencionados. Al dar solución a esta problemática, se tiene un punto de partida para crear una herramienta más robusta y con mayor variabilidad para generar pipelines de CI/CD y otras características que se percibieron para trabajos futuros.

Pontificia Universidad Javeriana Cali
Facultad de Ingeniería
Maestría en Ingeniería de Software
Proyecto de Grado

Creación Automática de Pipelines Para CI/CD Usando Técnicas de Reutilización y Variabilidad

Robinson Cruz Delgado

Director: Jaime Alberto Chavarriaga
Co-Director: Luisa Fernanda Rincón

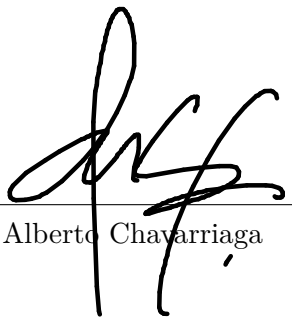
17 de Enero de 2025

Santiago de Cali, 17 de Enero de 2025

Señores
Pontificia Universidad Javeriana Cali.
Ph.D. Luisa Fernanda Rincón
Directora Maestría en Ingeniería de Software.
Cali.

Cordial Saludo.

Por medio de la presente hago constar que en mi calidad de director de trabajo de grado he revisado el proyecto titulado "Creación Automática de Pipelines Para CI/CD Usando Técnicas de Reutilización y Variabilidad" realizado por el estudiante de la Maestría en Ingeniería de Software Robinson Cruz Delgado (cod: 8969063), el cual se encuentra terminado y considero que cumple con los requisitos para ser entregado.
Atentamente,



Jaime Alberto Chavarriaga

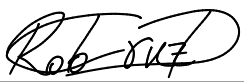
Santiago de Cali, 17 de Enero de 2025.

Señores
Pontificia Universidad Javeriana Cali.
Ph.D. Luisa Fernanda Rincón
Directora Maestría en Ingeniería de Software.
Cali.

Cordial Saludo.

Me permito presentar a su consideración el proyecto de grado titulado "Creación Automática de Pipelines Para CI/CD Usando Técnicas de Reutilización y Variabilidad" con el fin de cumplir con los requisitos exigidos por la Universidad y para que sea sometido a revisión del jurado y cumpla su aprobación, para conseguir posteriormente el título de Magíster en Ingeniería de Software.

Atentamente,



Robinson Cruz Delgado
Código: 8969063

0.1. Resumen

La integración y despliegue continuo, junto con los pipelines como código, han generado una revolución en la industria del software permitiendo publicar cambios desde el código en menos de una hora en producción. Sin embargo, los pipelines como código agregan los mismos problemas del código en general, como malas prácticas de desarrollo y configuración, vulnerabilidades, desconfianza o sobrecostos durante su ejecución. Aunque, se han creado herramientas para generar pipelines e intentar solventar estos problemas, algunos se enfocan en una o dos herramientas de CI/CD y no brindan variedad y en otros casos solo se llega el pipeline hasta la fase de pruebas o entrega continua, pero no hasta el despliegue. Por ello, se creó una herramienta que aproveche las cualidades de la variabilidad y la reutilización, para generar pipelines de CI/CD mitigando los problemas ya mencionados. Al dar solución a esta problemática, se tiene un punto de partida para crear una herramienta más robusta y con mayor variabilidad para generar pipelines de CI/CD y otras características que se percibieron para trabajos futuros.

Palabras Clave Integración continua, despliegue continuo, pipeline, reutilización de software, variabilidad de software

0.2. Abstract

Continuous integration and deployment, along with pipelines as code, have sparked a revolution in the software industry by enabling changes to be published from code to production in less than an hour. However, pipelines as code bring forth the same issues as code in general, such as poor development and configuration practices, vulnerabilities, lack of trust, or increased costs during execution. Although tools have been created to generate pipelines and attempt to address these issues, some focus on one or two CI/CD tools and lack variety, while in other cases, the pipeline only reaches the testing or continuous delivery phase, but not deployment. Hence, a tool was created to leverage the qualities of variability and reuse to generate CI/CD pipelines, mitigating the aforementioned problems. By addressing this issue, there is a starting point to create a more robust tool with greater variability to generate CI/CD pipelines and other features perceived for future works.

Keywords CI/CD, pipeline, generator, software reuse, software variability

Índice general

Índice de figuras

Índice de tablas

Capítulo 1

Introducción

En la actualidad la industria de software tiene una gran presión para entregar valor más rápidamente, introduciendo nuevas funcionalidades con mayor velocidad y sin afectar la continuidad de los servicios que ofrecen a sus clientes. Por ejemplo, empresas como Microsoft, Google y Amazon liberan constantemente nuevas funcionalidades de sus productos, varias veces al día, e incluso, varias veces por segundo [1].

Para lograr estos niveles de productividad, las empresas pueden utilizar esquemas de *Integración, Entrega y Despliegue continuo (CI/CD)*, a.k.a., procesos automatizados que permiten detectar defectos en el software (integración continua), generar artefactos que se pueden distribuir e instalar en los clientes (entrega continua) y desplegar las nuevas versiones sin generar ninguna interrupción en el servicio (despliegue continuo).

1.1. Pipelines de CI/CD

Los desarrolladores implementan los procesos de integración, entrega y despliegue continuos (CI/CD) usando uno o más *pipelines* por cada proyecto. Estos pipelines establecen las tareas que deben ejecutarse. Por ejemplo, estos pipelines pueden definir tareas como descargar el código fuente de los repositorios de código, construir el software, ejecutar pruebas de unidad, generar paquetes e instaladores y desplegar la aplicación en ambientes de pruebas y de producción (TechTarget, 2021).

Estos procesos normalmente se ejecutan usando herramientas como Jenkins[2], Github Actions[3], Azure Pipelines[4], Travis CI[5] y Circle CI[6]. En la actualidad, para estas herramientas los pipelines se especifican como parte del código fuente y se modifican y versionan como el resto del software (Labouardy, 2021).

Estos pipelines normalmente ejecutan otras herramientas para hacer el trabajo. Por ejemplo, considerando un escenario típico, un pipeline puede,

- Descargar el código fuente de algún repositorio, usando git.
- Construir una serie de librerías, por ejemplo, en Java o Javascript usando maven o npm

[1] https://www.allthingsdistributed.com/2014/11/apollo-amazon-deployment-engine.html
[2] https://www.jenkins.io/
[3] https://github.com/features/actions
[4] https://azure.microsoft.com/es-es/services/devops/pipelines/
[5] https://travis-ci.com/
[6] https://circleci.com/

- Construir una aplicación, por ejemplo, en Java usando maven.
- Ejecutar pruebas automatizadas usando maven y JUnit.
- Generar una imagen de contenedor usando Docker.
- Subir la imagen de contenedor usando Docker y algún registro de imágenes como DockerHub.
- Desplegar la aplicación en algún servicio de nube, como Azure App Services, usando algún comando de Azure CLI.
- Hacer pública la aplicación usando algún comando de comando de Azure CLI.

En términos generales, el pipeline define una serie de pasos que se pueden ejecutar con herramientas diferentes. Los desarrolladores deben conocer no solo la herramienta que permite ejecutar los pipelines, sino también todas las herramientas que se ejecutan como parte del proceso. Muchas veces, cuando se usan los mismos lenguajes y las mismas tecnologías, los pipelines terminan ejecutando casi siempre las mismas herramientas e incluyendo una serie de pasos muy similares.

Lamentablemente, en muchos casos, estos *pipelines* se construyen de forma manual y con técnicas de reutilización oportunista, por ejemplo, copiando y pegando código de archivos ya existentes, sin contar con técnicas de diseño y herramientas que faciliten la tarea.

1.2. Ejemplo: Pipelines para Aplicaciones web en Java

Considere, por ejemplo, el desarrollo y puesta en funcionamiento de una aplicación desarrollada en Java. Cada vez que se agrega una nueva funcionalidad o se corrige un error es necesario compilar el código, ejecutar pruebas unitarias, posiblemente desplegar la aplicación en un ambiente para la ejecución de pruebas manuales y, en caso que, no se tengan errores hacer el despliegue de la aplicación en un ambiente de producción. Estas tareas involucran diferentes herramientas: posiblemente, unas herramientas para compilar la aplicación, otra para ejecutar las pruebas y otras para hacer el despliegue. Los desarrolladores pueden construir un *pipeline* definiendo cada uno de los pasos del proceso y estableciendo las herramientas que ejecutan cada uno de los pasos.

Estos *pipelines* se pueden construir con herramientas como Github Actions [7]. En esta herramienta, los pipelines se definen usando un archivo en formato YAML (`.yml`) que define cuando se ejecuta el proceso, el tipo de ambiente en donde se debe ejecutar, las tareas a realizar y cada uno de los pasos. El listado 1.1 muestra un ejemplo. Las líneas 3 al 5 establecen que el pipeline debe ejecutarse cuando se hace un cambio en la rama `main`, la línea 10 indica que los procesos deben ejecutarse en un ambiente Ubuntu Linux y las líneas 12 al 24 definen una tarea `compilar-y-probar` que incluye pasos para descargar el código fuente, compilar el proyecto y ejecutar pruebas unitarias.

[7] https://docs.github.com/es/actions

```yaml
name: Compila aplicación Java usando Maven

on:
  push:
    branches: [ "main" ]

jobs:
  compilar-y-probar:

    runs-on: ubuntu-latest

    steps:
    - name: Descarga código fuente
      uses: actions/checkout@v4
    - name: Configura Java 17
      uses: actions/setup-java@v3
      with:
        java-version: '17'
        distribution: 'temurin'
        cache: maven
    - name: Compila el proyecto
      run: mvn -B package --file pom.xml
    - name: Ejecuta pruebas unitarias
      run: mvn test --file pom.xml
```

Listing 1.1: Ejemplo de pipeline en Github Actions

En estos *pipelines*, cada uno de los pasos (`steps`) puede utilizar una *acción* específica. En el listado 1.1, por ejemplo, el primer paso usa la acción `actions/checkout` para descargar el código fuente y el segundo paso usa la acción `actions/setup-java` para configurar la versión de Java a utilizar. La persona que construye el pipeline puede seleccionar alguna acción de un conjunto que existen en el Github Marketplace [8], leer la documentación y agregar nuevas lineas al pipeline especificando los parámetros apropiados de acuerdo con la tarea que se desea ejecutar. A principios de 2024 existían alrededor de 23,000 acciones diferentes disponibles en el marketplace.

Por ejemplo, existen acciones que permiten desplegar aplicaciones web en servicios de nube pública. La acción `azure/webapps-deploy` permite desplegar una aplicación web Java en la nube de Microsoft Azure usando el servicio de App Services. Si se desea desplegar una aplicación como parte de un pipeline, es posible agregar un paso usando el código mostrado en el listado 1.2.

Para desplegar la aplicación en otra nube pública o utilizando un servicio diferente, es necesario utilizar una acción diferente. Por ejemplo, la acción `azure/spring-apps-deploy` permite desplegar una aplicación web usando el servicio de Spring Apps de Microsoft Azure. El listado 1.3 muestra un ejemplo de código para desplegar una aplicación usando este servicio.

En nubes públicas como Microsoft Azure, la misma aplicación web puede funcionar utilizan-

[8] https://github.com/marketplace

```
1
2       - name: Despliega en Azure App Services
3         uses: azure/webapps-deploy@v2
4         with:
5           app-name: helloworld
6           publish-profile: ${{ secrets.AZURE_WEBAPP_PUBLISH_PROFILE }}
7           package: '${{ github.workspace }}/target/*.jar'
```

Listing 1.2: Ejemplo de acción para desplegar en Azure App Services

```
1
2       - name: Despliega en Azure Spring Apps
3         uses: azure/spring-apps-deploy@v1
4         with:
5           azure-subscription: ${{ env.AZURE_SUBSCRIPTION }}
6           action: Deploy
7           service-name: hello-service
8           app-name: helloworld
9           use-staging-deployment: false
10          package: ${{ github.workspace }}/**/*.jar
```

Listing 1.3: Ejemplo de acción para desplegar en Azure Spring Apps

do servicios diferentes. Dependiendo del servicio a utilizar, el desarrollador debe incluir acciones diferentes en los pipelines para ejecutar el despliegue correspondiente.

1.3. Ejemplo: Alternativas para despliegue en Microsoft Azure

Microsoft Azure ofrece varias alternativas para desplegar aplicaciones web desarrolladas en Java. Por ejemplo, una aplicación web desarrollada con el framework de Spring Boot puede funcionar en máquinas virtuales, en App Services, en Container Apps, Container Instances, Azure Kubernetes Services (AKS) o en Azure RedHat OpenShift (ARO), entre otros. Cada una de estas opciones tiene sus ventajas y desventajas. En muchos casos, las empresas empiezan usando un servicio y luego se migran a otro servicio a medida que sus necesidades cambian.

Existen varias guías [9] para seleccionar el servicio a utilizar para desplegar una aplicación web en Java. Por ejemplo, la figura 1.1 muestra un árbol de decisión propuesto por Microsoft. Allí se recomiendan servicios diferentes de acuerdo con diversos criterios. Se recomienda, por ejemplo, el servicio de App Services para aplicaciones que se van a migrar sin modificaciones (lift and shift) y que son desarrollos propios (no COTS). Igual, se recomiendan los App Services para aplicaciones nuevas o aplicaciones a migrar usando contenedores que requieren funcionalidades de hosting web

[9] https://learn.microsoft.com/en-us/azure/architecture/guide/technology-choices/compute-decision-tree

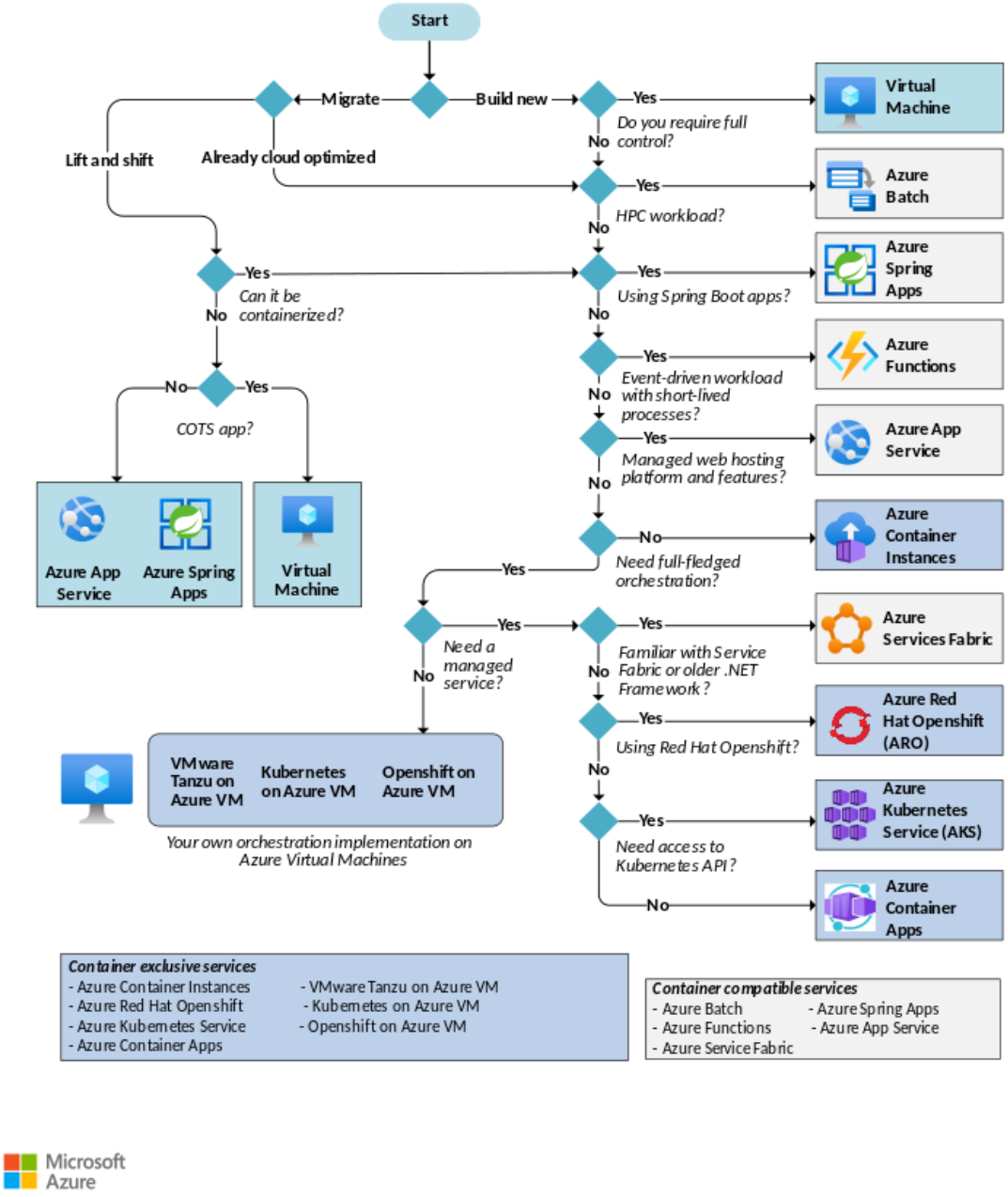


Figura 1.1: Árbol de decisión para seleccionar servicio de despliegue en Azure. Fuente, Microsoft Learn

administrado. Un desarrollador puede usar este árbol de decisión para seleccionar uno u otro servicio de acuerdo con sus necesidades.

Para las personas que construyen *pipelines* que hacen el despliegue de una aplicación, esta multitud de opciones representa varios retos al momento de construir un *pipeline* una vez se ha seleccionado un tipo de servicio y al momento de modificarlo cuando se decide cambiar el servicio.

1.4. Retos para el desarrollador

Como puede verse a partir de los ejemplos, la construcción de pipelines para integración y despliegue continuo trae consigo una serie de retos para los desarrolladores:

R1: *Dificultad de seleccionar las acciones*, una vez seleccionada la alternativa de despliegue. En muchos casos los desarrolladores deben buscar entre los más de 23 mil tipos de acciones, cuál es la más apropiada de acuerdo con la alternativa seleccionada. En ocasiones, el desarrollador selecciona una que no es apropiada o, luego de utilizar una, la cambia por otra que le ofrece una funcionalidad que necesita. Además, si el desarrollador no ha trabajado previamente con una acción específica, él debe revisar la documentación y los ejemplos para definir los parámetros a utilizar.

R2: *Dificultad de incluir estas acciones en los pipelines.* Sobre todo, cuando se cambia de un tipo de despliegue a otro, se hace necesario eliminar algunas acciones y agregar otras. Estas modificaciones pueden introducir errores o acciones innecesarias que un desarrollador puede no notar.

R3: *Dificultad de reutilizar conjuntos de acciones en los pipelines.* Sobre todo, en compañías con varios equipos de trabajo, es posible que las acciones y secciones completas de los pipelines sean diferentes, incluso cuando se usen las mismas tecnologías de despliegue. En muchos casos los desarrolladores solo hacen *copiar y pegar* de ejemplos en los sitios web sin emplear un mecanismo de reutilización apropiado. De hecho, al copiar y pegar porciones de código, esto lleva a tener código duplicado que aumenta el esfuerzo para corregir un error o hacer una optimización debido a que obliga a modificar el código en varios archivos diferentes.

1.5. Planteamiento del problema

Teniendo en cuenta los retos que enfrentan los desarrolladores para implementar pipelines de CI/CD, se plantea el siguiente interrogante:

- ¿Cómo implementar una herramienta que permita generar pipelines para diferentes herramientas de CI/CD, haciendo uso de técnicas de reutilización y variabilidad de código?

1.6. Sistematización

A continuación, se presentan las preguntas de investigación que ayudan a guiar la investigación y con la cual se responde a ellas.

- ¿Qué oportunidades de reutilización y manejo de variabilidad se pueden aplicar en la construcción de pipelines de CI/CD?
- ¿Cómo se puede implementar la generación automática de pipelines de CI/CD considerando las técnicas de reutilización y manejo de variabilidad identificadas previamente?
- ¿Qué ventajas y desventajas perciben los desarrolladores al utilizar los esquemas de generación automática que se propone en el presente proyecto?

1.7. Objetivos

1.7.1. Objetivo General

- Proponer un software que genere automáticamente pipelines para CI/CD variando las herramientas a usar utilizando técnicas de reutilización y variabilidad.

1.7.2. Objetivos Específicos

- Identificar y definir una técnica que permita reutilizar y variar el código de pipelines para CI/CD.
- Diseñar e implementar un software que genere automáticamente pipelines para CI/CD, mediante técnicas de reutilización y variabilidad.
- Demostrar el funcionamiento de la implementación usando proyectos de prueba como casos de estudio.
- Evaluar la satisfacción y utilidad de la solución planteada con personas que en su labor utilicen pipelines para CI/CD.

1.8. Justificación

La automatización de los procesos de integración, entrega y despliegue continuos en los pipelines de los proyectos de software, ha generado una gran velocidad para publicar en el mercado cambios y nuevas versiones, pero al tiempo vinculando una gran responsabilidad en la persona encargada de la gestión de estos pipelines, ya que, estos se convierten en la base para llevar el software desde el código hasta producción.

Adicionalmente, los pipelines introducen otro problema que afecta a estas personas encargadas de su desarrollo, porque al ser los pipelines otra pieza de código, conlleva los mismos problemas que las líneas de código de software tienen en general; como lo son, malas prácticas de programación, errores de configuración y vulnerabilidades en el software. Estos problemas anteriores, se pueden resolver automatizando la generación de estos pipelines, buscando la agilidad en su proceso de creación y los errores que puedan incorporar. Y aunque ya se han realizado proyectos previos que generan pipelines automáticamente, no se ha aprovechado en ellos técnicas de reutilización y variabilidad, lo cual le podría brindar otro porcentaje de eficacia a la generación y por tanto, mayor satisfacción y utilidad a la persona responsable de la gestión de pipelines y en el mismo camino al proyecto mismo por la eliminación de esos posibles problemas.

1.9. Delimitaciones y Alcances

Este trabajo de grado encuentra una nueva alternativa para la creación de pipelines para CI/CD que tienen las soluciones software. Teniendo como variables los sistemas en que estos se despliegan y la variedad de herramientas que se puede usar al crear estos proyectos de software en general. Para ello, se usa técnicas de reutilización y variabilidad en sistemas de despliegue de aplicaciones en nubes como Azure o AWS y en herramientas de CI/CD como Jenkins, Github Actions, TravisCI, CircleCI y Azure Pipelines, siendo unas de las más comunes.

La implementación se creó apoyándose de proyectos básicos de páginas web con una herramienta como Jhipster, y con la cual se generan los proyectos para el primer paso de evaluación. Posteriormente se valida la herramienta en factores como satisfacción y utilidad, con el apoyo de personas que en su trabajo tengan relación con pipelines de CI/CD o que tengan conocimiento de ellos.

1.10. Aspectos Metodológicos

1.10.1. Metodología de la Investigación

El enfoque metodológico adoptado para este trabajo de grado es el *Design Science Research* (DSR), descrito por vom Brocke et al. (2020). Dicha metodología se centra en la generación de conocimiento a través de la creación de artefactos innovadores, alineándose con los objetivos delineados en el proyecto. Estos objetivos buscan la generación automática de pipelines para CI/CD mediante técnicas de reutilización y variabilidad.

Siguiendo el modelo del proceso establecido para la metodología DSR, según vom Brocke et al. (2020), se detallan las siguientes actividades:

1. **Identificación del Problema y Motivación:** En esta etapa, se busca identificar claramente el problema a abordar, proporcionar la motivación y justificación para la realización del trabajo, así como establecer un marco teórico y revisión del estado del arte sobre el tema. Esta etapa se desarrolla durante el capítulo 1, 2 y 3.

2. **Definición de Objetivos para una Solución:** Esta actividad tiene como objetivo establecer los objetivos del trabajo, sirviendo como base para abordar el problema identificado en la actividad anterior. El desarrollo de esta etapa se da en la sección 1.7.

3. **Diseño y Desarrollo:** En esta fase, se trabaja en el desarrollo de los primeros tres objetivos específicos del proyecto. Se identifican las técnicas y herramientas a emplear, permitiendo diseñar una solución que se ajuste de manera óptima al problema planteado, culminando con su implementación. Esta fase se presenta en el Capítulo 4.

4. **Demostración:** Aquí se busca exhibir el funcionamiento de la implementación mediante el uso de proyectos de prueba como casos de estudio. La realización de esta etapa se presenta en el Capítulo 5 durante la fase 1.

5. **Evaluación:** Esta actividad implica la evaluación de la satisfacción y utilidad de la solución propuesta. Se llevará a cabo mediante pruebas con profesionales que utilicen pipelines para CI/CD, obteniendo retroalimentación a través de encuestas. Esta etapa se describe en el Capítulo 5, directamente en la fase 2 de este.

6. **Comunicación:** En esta fase se generará la monografía resultante del trabajo de grado.

1.10.2. Construcción de la Herramienta

El enfoque metodológico de Design Science Research propone un ciclo iterativo compuesto por tres tareas fundamentales: diseñar, implementar y evaluar, culminando con la fase de comunicación para difundir los resultados obtenidos (Tebes et al., 2019). Detalles más claros sobre el desarrollo de este proceso se encuentran descritos en el Capítulo 4 del presente documento. Este proporciona una visión integral del flujo de trabajo adoptado durante el diseño e implementación de los pipelines, seguido de la conceptualización y desarrollo del generador. A través de un enfoque detallado

y estructurado, se desglosa cada etapa del proceso, desde la identificación de requisitos hasta la implementación concreta de soluciones. Este enfoque facilita una comprensión completa de la evolución del proyecto. Estos temas sirven de preparación para la exploración posterior en el Capítulo 5, donde se examinan de manera rigurosa los diferentes procesos de evaluación llevados a cabo para validar la eficacia y utilidad de la herramienta desarrollada. Se exploran métricas, casos de uso y resultados obtenidos, brindando así una evaluación integral de la contribución del trabajo al contexto de CI/CD.

1.11. Contribuciones del proyecto

La principal contribución de este trabajo de grado es la introducción de una herramienta que permite generar automáticamente *pipelines* para CI/CD usando técnicas de reutilización y gestión de variabilidad de software. Específicamente,

C1: *Una revisión del estado del arte* en herramientas de generación de pipelines. Revisión que presenta beneficios y limitaciones de estas herramientas y que presenta opciones de mejora para futuras investigaciones sobre el tema.

C2: *Un modelamiento de la variabilidad para pipelines de CI/CD para despliegue* considerando aplicaciones web desarrolladas en Java sobre nube pública de Azure. Modelo que permite a los desarrolladores de herramientas definir estrategias de implementación para soportar cada una de las opciones y que, una vez se desarrolla una herramienta, permite soportar esquemas de configuración donde el desarrollador selecciona la opción que más le conviene de acuerdo con sus necesidades.

C3: *Una herramienta que genera automáticamente pipelines de CI/CD de acuerdo con la selección del usuario* extendiendo la herramienta JHipster, que permite la generación de aplicaciones web en Java. Esta implementación presenta una solución que puede ser utilizada por desarrolladores en sus labores diarias que puede ser una referencia para la construcción de nuevas soluciones.

1.12. Organización del documento

El presente documento está organizado en 6 capítulos: El Capítulo 2 presenta el marco de trabajo del proyecto describiendo las bases teóricas y los antecedentes del proyecto. El Capítulo 3 analiza algunas de las herramientas para la construcción de pipelines y describe la variabilidad que se presenta cuando se usan diferentes esquemas de despliegue en nube. El Capítulo 4 describe la selección de las herramientas y técnicas, al igual que el desarrollo del generador. El capítulo 5 presenta una evaluación de este utilizando dos fases de evaluación. Y finalmente el capítulo 6 presenta las conclusiones, lecciones aprendidas por el desarrollador y trabajos futuros.

Capítulo 2

Marco de referencia

Durante mucho tiempo, los desarrolladores construían el software a través de módulos que se probaban y se integraban solo al final del proceso de desarrollo. Además, el software solo se instalaba en el ambiente de producción cuando ya se había construido todo. En estos casos, muchos errores se detectaban muy cerca de la fecha de entrega, causando reprocesos y retrasos en el proyecto y frustración entre los desarrolladores (Duvall et al., 2007).

En la década de los noventa surgieron varias propuestas para integrar y desplegar el software continuamente, a veces varias veces al día, en lugar de hacerlo solo al final. Nuestro trabajo se enfoca en la construcción de *pipelines CI/CD*: los procesos automáticos que posibilitan esta integración y despliegue continuo (CI/CD). Este capítulo presenta las bases conceptuales y teóricas del proyecto.

2.1. Integración y Despliegue Continuo (CI/CD)

La **Integración Continua (CI)** inició en la década de los noventa como una práctica de software donde los diferentes módulos de software se integran y se prueban frecuentemente durante el transcurso del proyecto[1]. Esta práctica, definida inicialmente por Grady Booch para explicar la integración de software cuando se hace desarrollo iterativo (Booch, 1990), fue popularizada posteriormente como parte de metodologías ágiles como *Extremme Programming (XP)* (Beck, 1999), y *Lean Software Development* (Poppendieck and Poppendieck, 2003).

La **Integración y Despliegue Continuo (CI/CD)** surgió alrededor de 2010, extendiendo el proceso de Integración Continua para incluir tareas adicionales para la distribución del software a los clientes y el despliegue en ambientes de pruebas y producción (Humble and Farley, 2010). Según esta práctica, la construcción manual de paquetes que se pueden entregar a los clientes y la instalación manual del software son anti-patrones que deben evitarse (Humble and Farley, 2010).

En la práctica de CI/CD podemos identificar tres tipos de tareas que se ejecutan continuamente y se pueden automatizar con herramientas especializadas (TechTarget, 2021):

- **Integración Continua:** Donde se realiza una construcción continua del software, integrando y probando los diferentes módulos de software. El objetivo es poder detectar los problemas lo más temprano posible de forma que se pueda reducir el reproceso y el costo de solucionarlos.
- **Entrega Continua:** Donde se generan los paquetes, instaladores e imágenes de máquinas virtuales o contenedores que pueden ser validados y/o desplegados por las personas responsables. El objetivo es contar con un proceso estándar y predecible que permita garantizar la calidad de los entregables para los clientes.

[1] https://martinfowler.com/articles/continuousIntegration.html

- **Despliegue Continuo:** Donde se instala y/o se despliega el software en ambientes de pruebas, pre-producción o producción. Esto incluye prácticas de revisión, aprobación, actualización (*roll forward*) y recuperación en caso de fallos (*roll back*). El objetivo es poder hacer cambios en el software sin afectar la operación y la continuidad de negocio.

2.2. Pipelines de CI/CD

Los **Pipelines de CI/CD** son especificaciones del proceso y/o programas que automatizan los procesos de Integración y Despliegue Continuo (Humble and Farley, 2010). Estos programas responden a eventos del proceso de desarrollo y ejecutar diversas herramientas para construir, ejecutar pruebas automáticas y desplegar el software. En contraste con otros esquemas de construcción de software, tales como make o maven, los pipelines de CI/CD: (1) ejecutan sus procesos todos los días a una hora determinada o respondiendo a eventos tales como cuando un desarrollador actualiza el software y (2) típicamente encadenan la ejecución de otras herramientas, donde el resultado de ejecutar una de ellas puede determinar si se ejecutan algunas otras.

Las **Herramientas de CI/CD** permiten ejecutar los pipelines de CI/CD, i.e., los procesos de CI/CD, de forma automáticas. Entre las primeras herramientas podemos mencionar: CruiseControl (2001) [2], Hudson (2008) [3] y Jenkins (2011)[4]. En estas herramientas los pipelines se definían usando bloques o diagramas y se almacenaban en bases de datos y no como parte del código.

En la actualidad, muchos desarrolladores han reemplazado estas herramientas por servicios de cloud computing que soportan integración continua (Golzadeh et al., 2021). Entre estos servicios se pueden mencionar: TravisCI (2011) [5], Circle CI (2014) [6], AppVeyor (2014) [7], Gitlab CI (2015) [8] y Github Actions (2019)[9]. En estas herramientas, los pipelines no se almacenan en bases de datos si no que se manejan como si fuesen código fuente.

Pipelines como código (Labouardy, 2021) es una práctica en donde los pipelines de CI/CD son tratados como código fuente. Por un lado, los pipelines se especifican en archivos de texto que se incluyen en repositorios de código fuente, y por otro lado, los pipelines se construyen y modifican siguiendo el mismo proceso que se usa con el código fuente. Por ejemplo, en este esquema de pipelines como código, cualquier desarrollador puede solicitar la aprobación de los cambios que hay sobre los pipelines y los líderes técnicos pueden hacer la revisión por pares y hacer la aprobación correspondiente cuando se considere que cumplen con todos los criterios de calidad y adherencia a estándares de la compañía. Al final los pipelines son tratados como cualquier otro elemento de código fuente.

[2] https://web.archive.org/web/20180613010803/http://cruisecontrolnet.org/
[3] https://projects.eclipse.org/projects/technology.hudson
[4] https://www.jenkins.io/
[5] http://travis-ci.org/
[6] http://circleci.com/
[7] http://www.appveyor.com/
[8] http://docs.gitlab.com/ee/ci
[9] http://github.com/features/actions

Algunos autores han definido una serie de *consideraciones* al momento de diseñar los pipelines de CI/CD. Lachhman (2022) plantea las siguientes consideraciones:

- **Calidad, cobertura y automatización de pruebas:** al diseñar los pipelines es necesario revisar el tipo de pruebas y la cobertura de las pruebas automatizadas que se ejecutan.
- **Seguridad:** incorporar controles y herramientas de seguridad para revisar que las aplicaciones sean seguras.
- **Configuración del entorno de destino:** automatizar el entorno destino del despliegue de forma que se pueda garantizar que esté listo para la aplicación.
- **Variables y secretos:** proteger las credenciales y otros datos sensibles para que no sean visibles para usuarios no autorizados.
- **Estrategia de lanzamiento:** considerar el patrón de despliegue de la aplicación, por ejemplo, azul/verde o *canary*, utilizado en el pipeline.
- **Aprobaciones:** requerir aprobaciones antes de realizar cambios en la infraestructura o de hacer un despliegue.
- **Auditoría y cumplimiento:** contar con controles y registros de auditoría que permitan cumplir con las normas y regulaciones aplicables.
- **Estrategias de reversión y fallas:** contar con estrategias de reversión y/o recuperación ante fallas.
- **Acuerdos de Nivel de Servicio (ANSs, ONSs e INSs):** contar con controles que permitan asegurar que los sistemas, entornos y servicios funcionen cumpliendo con los acuerdos establecidos.

2.3. Herramientas de Integración y despliegue continuo

Las herramientas de CI/CD permiten administrar y ejecutar los pipelines definidos para diferentes productos y proyectos de desarrollo. Entre estas herramientas podemos mencionar:

- **Jenkins:** herramienta de código abierto y gratuita que ofrece diversas opciones para ejecutar pipelines CI/CD[10]. Puede utilizarse en entornos híbridos, en infraestructura del cliente o usando servicios de nube.

  Cabe mencionar que Jenkins tiene dos tipos de sintaxis para construir los pipelines, "declarative" y "scripted". según la documentación, la sintáxis declarativa es la opción recomendada debido a que es más fácil de entender y, si se requiere, puede incluir scripts en su funcionamiento[11].

[10] https://www.jenkins.io/doc/
[11] https://www.jenkins.io/doc/book/pipeline/

- **Azure Pipelines:** herramienta ofrecida por Microsoft como parte de Azure DevOps y con un alto nivel de integración con otros servicios de Azure[12]. Esta herramienta requiere del uso de un servicio en nube y permite ejecutar las tareas de los pipelines en nube, en infraestructura privada o en los computadores de los desarrolladores.
- **Github Actions:** herramienta integrada a la plataforma de alojamiento de código GitHub[13]. Requiere del uso de un servicio en nube y permite ejecutar las tareas de los pipelines en nube, en infraestructura privada o en los computadores de los desarrolladores.
- **CircleCI:** herramienta ofrecida como servicio en nube[14].
- **TravisCI:** herramienta ofrecida como servicio en nube[15].
- **Gitlab CI/CD:** herramienta integrada a la plataforma de alojamiento de código GitLab[16]. Requiere del uso de un servicio en nube y permite ejecutar las tareas de los pipelines en nube, en infraestructura privada o en los computadores de los desarrolladores.

La selección de la herramienta de CI/CD adecuada depende de las necesidades específicas de cada proyecto y equipo. Algunos factores por considerar incluyen:

- **Tamaño y complejidad del proyecto:** Las herramientas como Jenkins, Azure Pipelines y GitLab CI/CD son adecuadas para proyectos grandes y complejos debido a sus diferentes opciones para ejecutar tareas en nube o en infraestructura local, siendo adecuadas para proyectos grandes y complejos. Además, Azure Pipelines incluye opciones para especificar diferentes entornos de despliegue y contar con flujos de aprobación como parte del proceso. Otras herramientas, como Travis CI y CircleCI, no ofrecen estas opciones y pueden ser más adecuadas para proyectos pequeños y medianos.
- **Experiencia del equipo:** Jenkins tiene un mayor tiempo en el mercado y existe una gran cantidad de profesionales con experiencia en él. Herramientas como Azure Pipelines y GitHub Actions tienen una mejor integración con Azure DevOps y Github y pueden ser más apropiadas para equipos que ya usen esas tecnologías.
- **Integración con otras herramientas:** Cada herramienta cuenta con una interfaz y una serie de componentes que permiten integrar los pipelines con otras herramientas. Por ejemplo, Azure DevOps y GitHub tiene una gran cantidad de plugins y cuentan con un buen nivel de integración con Azure. Dependiendo de las tecnologías que use el equipo de desarrollo, es posible que una herramienta específica sea más apropiada de acuerdo con el nivel de integración que tiene.

[12] https://learn.microsoft.com/en-us/azure/devops/pipelines/
[13] https://docs.github.com/actions
[14] https://circleci.com/docs/
[15] https://docs.travis-ci.com/
[16] https://docs.gitlab.com/ee/ci/

- **Presupuesto:** Algunas herramientas, como Jenkins, son de código abierto y gratuitas. Otras herramientas, como Azure DevOps y Github, aunque ofrecen planes gratuitos, normalmente requieren el pago de una suscripción cuando se requieren algunas funcionalidades o se requieren ejecutar muchos pipelines como parte del mismo proyecto.

2.4. Reutilización y Variabilidad en el Software

La **Reutilización de Software** busca reducir el tiempo de desarrollo y los costos de producción y mantenimiento mediante la construcción y el uso del mismo código fuente, librerías y/o componentes en varios productos y proyectos de desarrollo de software (Sommerville, 2016). La reutilización de software se puede dar en diferentes niveles, como pueden ser sistemas completos, aplicaciones, componentes u objetos y funciones.

Aunque la reutilización de software permite acelerar el desarrollo y reducir costos, normalmente requiere de la construcción de componentes y servicios especialmente diseñados para ser reutilizados. Además, en muchas ocasiones, la reutilización implica tareas en donde los desarrolladores deben buscar y seleccionar cuál es el componente o servicio más apropiado para su proyecto específico. En algunos casos, un equipo de desarrollo puede preferir construir algunas funcionalidades de software de nuevo, en lugar de utilizar un componente o servicio ya existente. La Tabla 2.1 presenta algunos beneficios y limitaciones de la reutilización de software:

Beneficios	**Problemas**
Desarrollo acelerado	Creación, mantenimiento y uso de una biblioteca de componentes
Uso efectivo de especialistas	Encontrar, comprender y adaptar componentes reutilizables
Mayor confiabilidad	Aumento de los costos de mantenimiento
Menores costos de desarrollo	Falta de soporte de herramientas
Reducción del riesgo de procesos	Síndrome de “no inventado aquí”
Cumplimiento de estándares	

Tabla 2.1: Beneficios y problemas de reutilización de software.

Existen diversas **técnicas para la reutilización de software**, i.e., técnicas para construir los componentes y servicios reutilizables y para construir nuevas aplicaciones a partir de esos componentes y servicios. Por ejemplo, existen *técnicas de reutilización basadas en patrones*, donde se utilizan patrones de diseño (Gamma et al., 1994) para construir librerías de clases y frameworks que pueden utilizarse en varias aplicaciones.

Entre las diferentes técnicas y enfoques para el reuso de software, podemos mencionar (Sommerville, 2016):

- **Application Frameworks:** Colecciones de clases abstractas y concretas que se pueden adaptar y extender para crear sistemas de aplicación.

- **Librerías:** Bibliotecas de clases y funciones que implementan abstracciones comúnmente utilizadas.
- **Componentes:** Elementos reutilizables que ofrecen una interfaz y cumplen con algún modelo de componentes, permitiendo ser utilizados sin conocer su implementación o tener acceso a su código fuente.
- **Servicios:** Sistemas y componentes que se ejecutan de forma independiente, por fuera de la aplicación y que pueden ser accedidos por un interfaz de programación normalmente a través de protocolos de red.
- **Aspectos:** Funcionalidades comunes que se pueden incluir (entrelazar o tejer) en diferentes clases y lugares de la aplicación.
- **Sistemas configurables:** Sistemas específicos de dominio que pueden configurarse según las necesidades específicas de un cliente o de un proyecto.
- **Lineas de productos de software:** Conjuntos de aplicaciones que se diseñan a partir de un esquema de arquitectura común que puede adaptarse a diferentes clientes. Normalmente, la solución se configura de acuerdo con las necesidades de cada cliente y el software utiliza uno y otro componente o servicio de acuerdo con la configuración suministrada.
- **Ingeniería basada en modelos:** Esquema en donde el software se representa como modelos de dominio y el código se puede generar automáticamente a partir de estos modelos.
- **Generadores de programas:** Sistemas que permiten generar el código de las aplicaciones a partir de una especificación, un lenguaje específico de dominio o un modelo del software requerido.

Para diseñar y construir las soluciones reutilizables, los desarrolladores pueden hacer un *análisis de dominio* en donde se determinen cuáles son todas las opciones y variaciones que debe soportar cada componente o servicio a desarrollar.

La **Variabilidad** se define como la habilidad de un software o componente de estar sujeto a variación van Gurp et al. (2002), i.e, de variar su funcionamiento de acuerdo a una configuración suministrada por el usuario y/o a las condiciones del entorno. Esta técnica se enfoca primero en definir los dominios de variedad que existen en los componentes a construir. Por ejemplo, Renault (2014), para construir sus vehículos, describe la variabilidad en relación con diferentes aspectos de los automóviles, como lo son, la diversidad de demanda de los clientes respecto a motor, tamaño de cabina, color, entre otros. Otras variaciones se dan por las regulaciones que tiene cada país para los autos que circulan en ellos, al igual que el clima y la infraestructura que estos tienen. Según lo descrito, por ejemplo, en el caso de Renault estas variaciones implican que los productos de vehículos deban soportar las diferentes variaciones en cada uno de estos dominios.

Capítulo 3

Estado del arte

La construcción de pipelines de CI/CD ha sido motivo de múltiples investigaciones en los últimos años. Por ejemplo, existen estudios sobre los problemas que enfrentan los desarrolladores al construir *pipelines*, sobre procedimientos automatizados para construirlos, para detectar problemas en ellos, y para optimizar su construcción y ejecución.

Este capítulo presenta una revisión del estado del arte relacionado con la generación automatizada de pipelines de CI/CD y la implementación de técnicas de reutilización en la construcción de estos pipelines, temas de especial atención para nuestro proyecto.

3.1. Preguntas Orientadoras

Para llevar a cabo nuestra revisión del estado del arte se definieron una serie de preguntas orientadoras:

P01: *¿Qué herramientas existen para generar pipelines de CI/CD de forma automática?*

P02: *¿Cuáles son las principales características de estos generadores de pipelines de CI/CD?*

P03: *¿Qué limitaciones tienen estos generadores de pipelines de CI/CD?*

3.2. P01: ¿Qué herramientas existen para generar pipelines de CI/CD de forma automática?

Para determinar las herramientas que permiten generar pipelines de CI/CD de forma automática, se realizó una revisión de literatura acorde con las guías de Kitchenham and Charters (2007).

3.2.1. Estrategia de Búsqueda

El primer paso consistió en buscar los documentos relacionados con nuestras preguntas orientadoras. Para ello, (1) definimos una cadena de búsqueda utilizando palabras en inglés relacionadas con la generación automática de pipelines para integración continua, entrega continua y despliegue continuo; y (2) utilizamos esa cadena para buscar los documentos relacionados en bases de datos bibliográficas.

Bases de datos bibliográficas: La búsqueda se realizó en marzo de 2024 usando Google Scholar, ResearchGate, IEEE Xplore y Science Direct.

Cadena de búsqueda: Para cada una de las bases de datos fue necesario crear una cadena de búsqueda diferente. Independiente de la base de datos a utilizar, la cadena de búsqueda se determinó de acuerdo con los siguientes criterios:

- **Título:** El título debe incluir *generation*, *generator*, *create*, *automatic*, *automation* o *automated*, seguido de *pipeline* o *pipelines*.
- **En todo el documento:** adicionalmente, en cualquier parte del documento deben aparecer *continuous*, *integration*, *delivery*, *software*, *deploy* o *deployment*, con el fin de enfocar los documentos a aquellos que tratan temas de software y de CI-CD.

Por ejemplo, para la búsqueda en Google Scholar la consulta se realiza de la siguiente forma:

(intitle:“generation” OR intitle:“generator” OR intitle:“create” OR intitle:“automatic” OR intitle:“automation” OR intitle:“automated”) AND (intitle:“pipeline” OR intitle:“pipelines”) AND “continuous” AND “integration” AND “software” AND “delivery” AND (“deploy” OR “deployment”)

Y, para IEEE Xplore, la consulta se forma de la siguiente manera:

(((“Document Title”:generation) OR (“Document Title”:generator) OR (“Document Title”:create) OR (“Document Title”:automatic) OR (“Document Title”:automation) OR (“Document Title”:automated)) AND ((“Document Title”:pipeline) OR (“Document Title”:pipelines)) AND (“All Metadata”:continuous) AND (“All Metadata”:integration) AND (“All Metadata”:software) AND (“All Metadata”:delivery) AND ((“All Metadata”:deploy) OR (“All Metadata”:deployment)))

A partir de la búsqueda, identificamos 54 artículos luego de remover todos los duplicados.

3.2.2. Identificación de Estudios Relevantes

En una primera etapa se determinaron cuáles eran estudios relevantes, relacionados directamente con nuestras preguntas orientadoras.

Criterios de Inclusión: Establecimos, como criterios de inclusión y exclusión que el artículo debe (1) estar en idioma inglés o español; (2) describir la creación de una herramienta, framework o sistema que genere pipelines para una o más herramientas de CI/CD; y (3) haber sido publicado después del año 2018.

En primera instancia se analizaron los artículos considerando su título y resumen y se descartaron 38 artículos. Los demás artículos fueron leídos por completo y se excluyeron 11 artículos usando los mismos criterios. Al final, se obtuvieron 5 estudios relevantes que se incluyeron en el análisis de la literatura. la figura 3.1 presenta gráficamente este proceso.

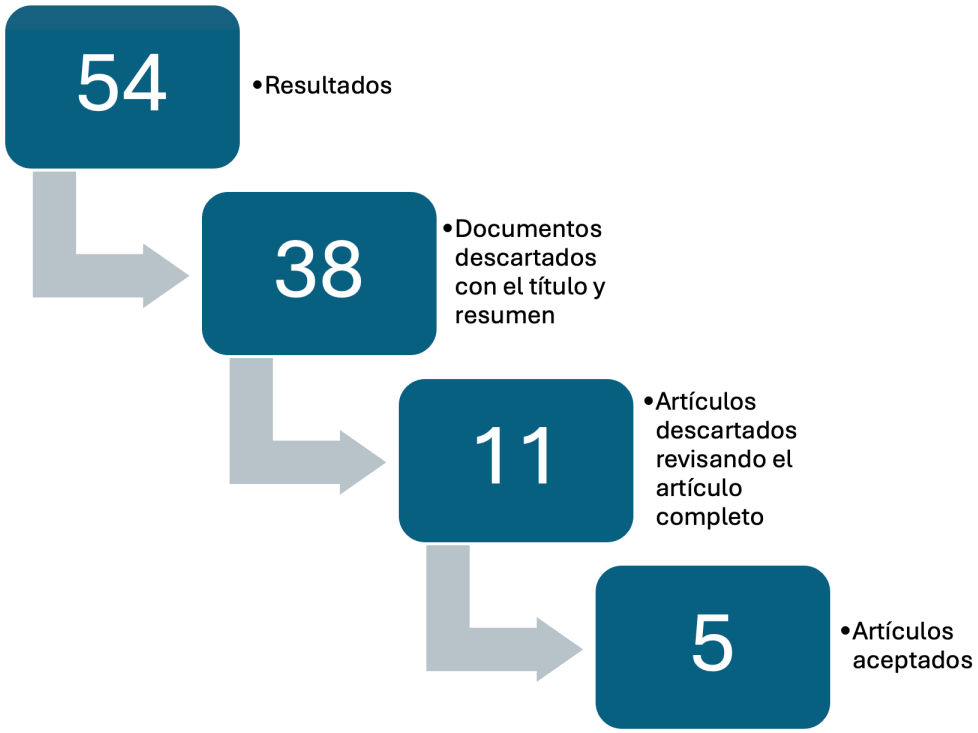


Figura 3.1: Búsqueda de estudios relacionados. Fuente propia

3.2.3. Resultados y Discusión

La Tabla 3.1 presenta el listado de artículos revisados:

Autores	Título	Año
Donca et al.	Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects	2022
Djounang-Nana	Towards the Automatic Identification of Optimal Configurations of CI/CD Pipelines of Software Development Projects: Symbolic, Meta-heuristic and Statistical Approaches	2022
Laaja	Design of an automated pipeline to improve the process of cross-platform mobile building and deployment	2022
Jones	Using Code Generation to Enforce Uniformity in Software Delivery Pipelines	2018
Aydin et al.	Automated Construction of Continuous Delivery Pipelines from Architecture Models	2021

Tabla 3.1: Listado de artículos revisados

A continuación, se incluye un resumen de cada uno de ellos.

3.2.3.1. Method for Continuous Integration and Deployment Using a Pipeline Generator for Agile Software Projects

Donca et al. (2022) presentan una herramienta para generar automáticamente pipelines de CI/CD a partir de los archivos de compilación y despliegue que encuentra en el repositorio Git del proyecto[1]. Cada vez que los desarrolladores integran algún cambio en el repositorio, cuando se ejecuta un *Merge Request*, la herramienta utiliza una serie de funciones y scripts para mantener actualizado los pipelines del proyecto.

Cuando un desarrollador realiza un *Merge Request* la herramienta determina si se han hecho cambios en los archivos de compilación o de configuración del proyecto. Cuando estos archivos han sido modificados, la herramienta genera los archivos del pipeline, i.e., el archivo `.gitlab-ci.yml` y una serie de archivos en la carpeta `/.ci` en el repositorio, envía estos cambios al repositorio Git y cancela la ejecución del pipeline actual. Debido a que la herramienta genera un nuevo cambio en el repositorio, Gitlab ejecuta el pipeline de nuevo, pero esta vez usando los archivos modificados.

En la práctica, todos los archivos del pipeline se actualizan constantemente a medida que los desarrolladores van construyendo el software. La única excepción es el archivo `.ci/custom.yml`, que puede contener tareas personalizadas.

La herramienta genera los pipelines usando una serie de tareas o *jobs* por defecto. Estos jobs incluyen acciones comunes como el versionado del proyecto, la construcción de imágenes Docker, el almacenamiento de las imágenes en un registro en AWS ECR y el despliegue de charts de Helm en Kubernetes. Para agregar otras características al pipeline, los desarrolladores pueden definir jobs personalizados agregando instrucciones al archivo `.ci/custom.yml`.

La Figura 3.2 describe las tareas (jobs) por defecto de un pipeline generado por la herramienta.

En este pipeline se pueden identificar algunas tareas de integración continua, donde se versiona, construye y publica la imagen de contenedor, y otras tareas de despliegue continuo, donde se versiona nuevamente y se despliega la aplicación usando Helm. Entre estas tareas podemos mencionar,

- *tarea de versionado*, cuando un cambio se fusiona (se hace un *merge*) en la rama principal de Git, se bloquean todos los pipelines sobre el mismo repositorio hasta que el versionado esté completo. Si el commit contiene cambios de código desplegables, se marca como candidato a versión (*release candidate*), y se libera el bloqueo. Posteriormente, se ejecutan las pruebas y, si todas las pruebas pasan, se marca el como una versión estable. En la figura 3.3 se describe este proceso.

- *tarea de compilación y creación de imagen Docker* que crea una imagen de contenedor a partir de los archivos de compilación y de "Dockerfile" en la carpeta raíz del proyecto. Primero, la tarea etiqueta la imagen de contenedor usando el nombre del proyecto como un prefijo, el nombre de la rama y el identificador hash del commit. Luego, esta imagen se publica en un registro "Elastic Container Registry" de AWS. La herramienta soporta la existencia de varios archivos Dockerfile en el repositorio y realiza los mismos pasos para cada uno de ellos.

[1] `https://www.mdpi.com/1424-8220/22/12/4637`

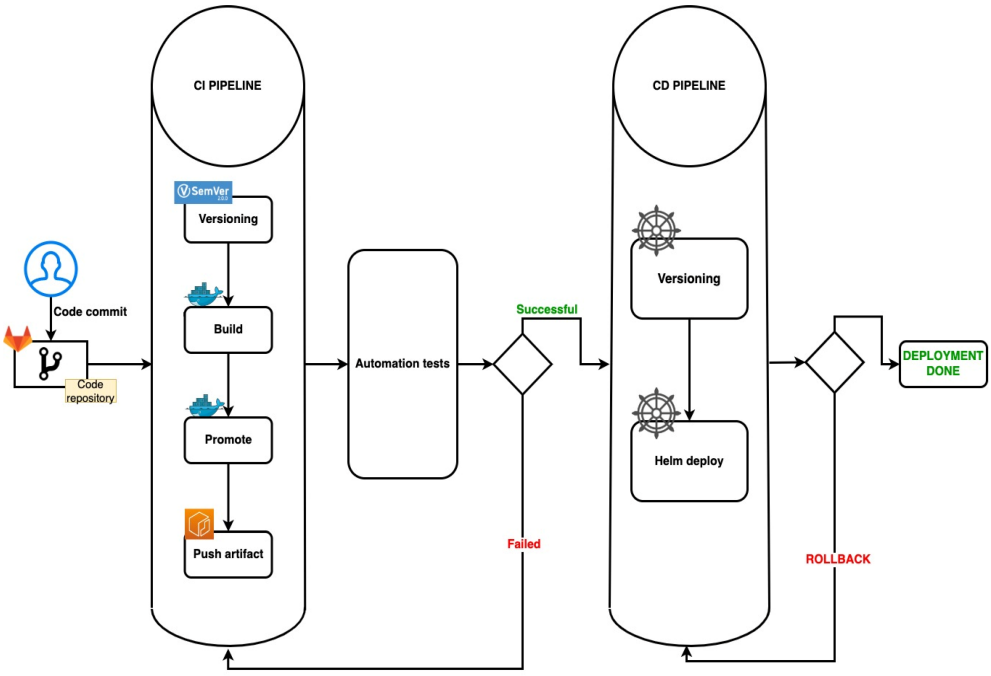


Figura 3.2: Diagrama de flujo de un pipeline. Fuente (Donca et al., 2022)

- *tarea de despliegue continuo* despliega las nuevas versiones del software en Kubernetes usando Helm y actualiza la información del despliegue en el archivo `Chart.yaml`. Si el despliegue falla, se activa un job de rollback que retorna la aplicación a la versión estable anterior.

3.2.3.2. Towards the Automatic Identification of Optimal Configurations of CI/CD Pipelines of Software Development Projects: Symbolic, Meta-heuristic and Statistical Approaches

Djounang-Nana (2022) presenta una versión preliminar de una herramienta para generar pipelines con configuraciones óptimas para la herramienta CI/CD donde se va a ejecutar[2]. Esta herramienta permite, por ejemplo, que los desarrolladores puedan migrar sus pipelines de Github Actions a otra herramienta generando pipelines optimizados para esta nueva plataforma. Para lograr ese objetivo, la herramienta cuenta con una serie de modelos que representan los pipelines que se pueden implementar en las diferentes herramientas y variabilidad en esos pipelines. Para generar un pipeline, se propone que la herramienta primero analice los diferentes elementos del proyecto y determine los pasos que debe ejecutar el pipeline, luego optimice el proceso usando alguna de tres estrategias: el uso de meta-heurísticas, de optimización simbólica o de optimización estadística, y finalmente, genere el código del pipeline usando un traductor de tareas.

[2] https://icsr2022v2.wp.imt.fr/files/2022/06/ICSR_DS_2022_DjounangNana.pdf

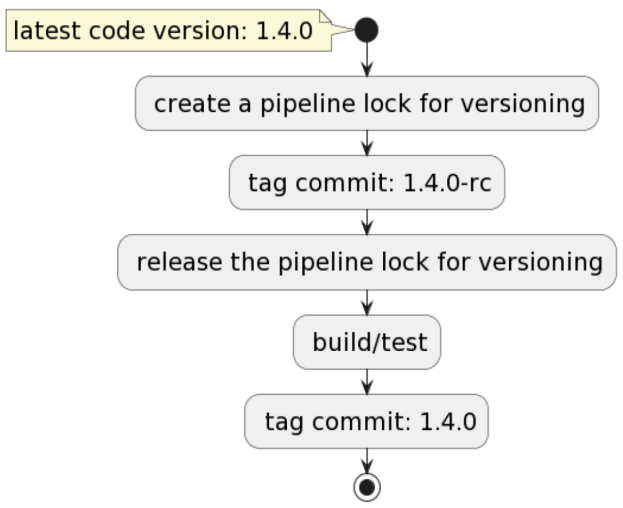


Figura 3.3: Descripción del flujo de versionado. Fuente (Donca et al., 2022)

En este artículo, los autores presentan un análisis de dominio de las herramientas de CI/CD. Presentan unos metamodelos que representan la sintaxis para implementar pipelines en plataformas como CircleCI, GitHub Actions y, con una mayor profundidad, GitLab CI. Los autores plantean que estos metamodelos pueden consolidarse en un metamodelo común, que soporte todas las plataformas, pero este metamodelo no se presenta en el documento.

3.2.3.3. Design of an automated pipeline to improve the process of cross-platform mobile building and deployment

Laaja (2022) presentan un generador de pipelines de CI/CD para publicar aplicaciones móviles Android y iOS en las tiendas Google Play y Apple Store respectivamente, usando GitLab como plataforma de CI/CD[3].

Para las aplicaciones móviles el proceso de despliegue tiene ciertas consideraciones diferentes a las que se encuentran en los procesos para desplegar aplicaciones web: por un lado, cada una de las tiendas de aplicaciones móviles tiene restricciones y pasos diferentes que deben tenerse en cuenta. Por otro lado, aunque el proceso de construcción no es muy complicado, normalmente requiere de una serie de pasos manuales y el uso de computadores Apple con OSX. Esto resulta en la posibilidad de introducir errores humanos en el proceso. Al automatizar todo el proceso se minimiza este riesgo. La Figura 3.4 presenta un diagrama de secuencia que describe el proceso manual para publicar las aplicaciones en un entorno de producción.

La implementación del pipeline se divide en varios pasos: preconfiguración tanto de Gitlab como de las máquinas de compilación, versionado, compilación y publicación de las aplicaciones en cada

[3] https://helda.helsinki.fi/server/api/core/bitstreams/848f7be7-d6c8-49ac-9c59-4c0e85b8a0d6/content

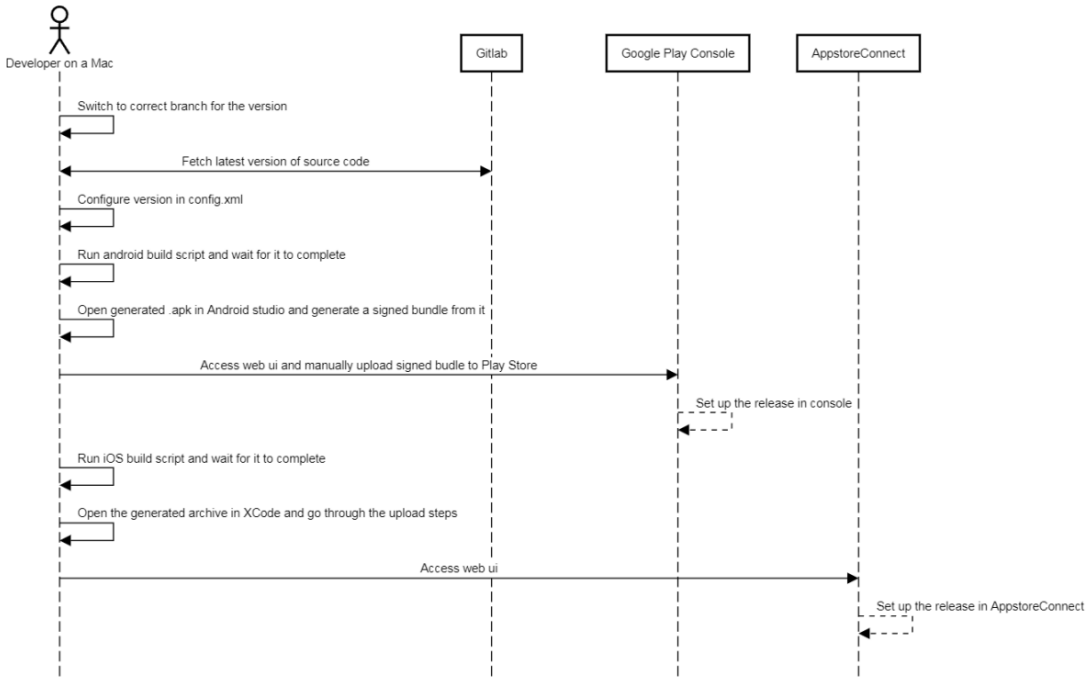


Figura 3.4: Flujo de despliegue manual de aplicaciones móviles a un entorno de producción. Fuente (Laaja, 2022)

tienda, y almacenamiento de datos de ejecución en el repositorio Git.

- **Preconfiguración** requerida para poder construir y desplegar las aplicaciones. Por ejemplo, para construir una aplicación Ionic que usa servicios de Firebase, se requieren configuraciones para Cordova (en archivo `config.xml` y `build.json`) y la descarga de archivos de configuración desde el portal de administración d Firebase (`google-services.json` para Android y `GoogleService-Info.plist` para iOS). Estas configuraciones normalmente deben incluirse en el código fuente en el repositorio, como secretos en la plataforma de CI/CD, Gitlab en este caso, en las máquinas usadas por los pipelines para compilar las aplicaciones, en uno o varios de estos sitios.

- **Versionado**, en este caso la información de la versión, número de compilación de Android y número de compilación de iOS, que debe verse reflejado en los diferentes archivos de código fuente y de configuración de la compilación. Por ejemplo, esta información debe reflejarse en archivos `build.xml` para Cordova y `package.json` para npm. Para hacer este versionamiento, la herramienta genera un pipeline que ejecuta instrucciones `sed` para reemplazar los datos de la versión en todos los archivos requeridos. Además, el pipeline generado actualiza el código fuente en el repositorio Git para conservar los cambios hechos en las versiones.

- **Construcción**, que involucra pasos relacionados con la descarga y configuración de librerías, la compilación del código fuente y la publicación en tienda de las aplicaciones. En el caso particular de aplicaciones móviles multiplataforma en Ionic, la construcción se realiza usado Cordova y requiere de la descarga y configuración de librerías con funcionalidades nativas. Además, luego de generar la aplicación, es necesario optimizar los binarios, firmar los archivos y ejecutar algunos comandos para publicar la aplicación en la tienda.
- **Actualización del repositorio**, a través de commits que actualizan la información de las versiones y de las configuraciones realizadas por el pipeline. En la herramienta propuesta este proceso incluye la descarga completa del repositorio y una serie de revisiones adicionales para garantizar que este proceso no falla y no se pierde la información resultante del pipeline.

La Figura 3.5 muestra el proceso automático que realiza el pipeline propuesto. Allí se observa una gran reducción en la interacción humana en comparación con el proceso inicial descrito en la Figura 3.4.

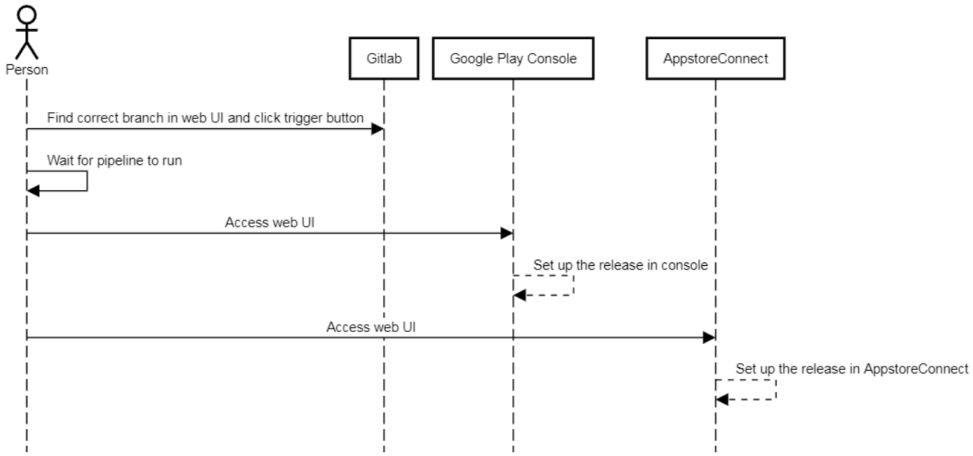


Figura 3.5: Flujo de despliegue automático a producción Apps. Fuente (Laaja, 2022)

3.2.3.4. Using Code Generation to Enforce Uniformity in Software Delivery Pipelines

Jones (2018) presenta la herramienta SPaaS (*software pipelines as a Service*) que genera pipelines para Jenkins y AWS CodeSuite que utiliza un lenguaje específico de dominio (DSL) para establecer las tareas que debe realizar y que genera el código de los pipelines usando plantillas en freemarker [4]. Esta herramienta surgió como respuesta a la necesidad de migrar más de 300 aplicaciones que una empresa desplegaba usando herramientas de CI/CD como CloudBees Enterprise Jenkins Server e IBM UrbanCode Deploy (uDeploy) a las herramientas de AWS CodeSuite.

[4] https://link.springer.com/chapter/10.1007/978-3-030-06019-0_12

En esta herramienta, los pipelines se definen usando un lenguaje específico de dominio (un DSL) basado en fases, actividades y scripts de eventos:

- **Fases o Etapas** (*phases* o *stages*), establecen grupos de actividades que deben ejecutarse secuencialmente. Cada fase solo se puede ejecutar luego de ejecutar las fases previas. El lenguaje específico de dominio establece cinco fases predefinidas a partir de las definiciones de Humble and Farley (2010): las fases de `commit`, `functional testing`, `capacity testing`, `acceptance testing` and `production`. La herramienta no permite definir fases adicionales y, por tanto, para crear nuevas fases sería necesario modificar el lenguaje específico de dominio y el código fuente de la herramienta.

- **Actividades** (*activities*), define tareas específicas dentro de cada fase. La herramienta incluye una serie de actividades predefinidas que se ejecutan si el pipeline especifica esta actividad y provee los parámetros necesarios. Las actividades además definen *eventos* que pueden ser procesados por scripts adicionales provistos por el desarrollador del pipeline.

- **Scripts de eventos** (*event scripts*), son scripts que se pueden ejecutar en respuesta a eventos dentro del pipeline. Por ejemplo, el artículo menciona que la actividad *build* genera un evento `onBuild` y la actividad *setup* genera un evento `onSetup`. Al final, el desarrollador puede incluir pasos para un proyecto específico definiendo scripts que se invoquen en respuesta a estos eventos. La herramienta soporta scripts de eventos como scripts *jenkinsfile* que pueden correr en un agente de ejecución de Jenkins o como imágenes de Docker que se pueden ejecutar en contenedores.

Adicionalmente, cada pipeline puede incluir propiedades, compuertas de decisión y aprobaciones.

- **Propiedades** (*Properties*), tuplas de nombre-valor que permiten al desarrollador especificar variables y parámetros a utilizar en los pipelines. Estas propiedades pueden definirse en diferentes *scopes*: se pueden definir a nivel del pipeline, a nivel de una fase o a nivel de una actividad. Esto significa que una propiedad definida a nivel de propiedad puede sobreescribir, para esa actividad específica, propiedades ya definidas a nivel del stage o del pipeline. Al final, al momento de ejecutar cada actividad, los programas reciben estas propiedades como variables de entorno.

- **Compuertas** (*Gates*), condiciones que deben ser satisfechas para ejecutar un pipeline, una etapa o una actividad. Si la condición no es satisfecha, las actividades correspondientes se ignoran y el resto del pipeline continúa su ejecución. El DSL define dos tipos de compuertas: *skip*, que define condiciones para omitir la ejecución de las actividades y *branch*, que define cuáles ramas del repositorio git deben iniciar su ejecución .

- **Aprobaciones** (*Approvals*), solicitudes de aprobación que deben ser atendidas por un usuario para continuar con la ejecución de actividades del pipeline. Cada aprobación puede definir un *timer* que establece la cantidad de tiempo que se debe esperar por la aprobación antes

de generar un fallo y un approver que define el nombre del usuario específico o del grupo de usuarios que puede realizar la aprobación.

Para utilizar la herramienta en un proyecto de software cualquiera, los desarrolladores deben definir un archivo `pipeline.dsl` que contiene las definiciones del pipeline. Luego, pueden usar la herramienta SPaaS para tomar esta definición y generar los archivos de código fuente del pipeline correspondiente. La herramienta produce el archivo del pipeline combinando el código prestablecido para las fases y actividades con el código provisto por los desarrolladores para los scripts de eventos.

Según los autores, los principales beneficios de la herramienta consisten no solo en la facilidad de migración de los pipelines de una herramienta de CI/CD a otra, sino también en la definición de un esquema genérico de pipelines, en lugar de pipelines completamente distintos para cada proyecto. Este esquema genérico posibilita la reutilización de las actividades de pipelines y el seguimiento de políticas de seguridad y de ejecución que se apliquen para todos los pipelines.

3.2.3.5. Automated Construction of Continuous Delivery Pipelines from Architecture Models

Aydin et al. (2021) presenta una herramienta, llamada JARVIS, para la construcción de pipelines basados en la definición de la arquitectura del software a desplegar, es decir, usando pipelines basado en la especificación de los artefactos (*artifact-based*), en lugar de usar pipelines basados en las actividades de compilación y despliegue (*activity-based*)[5].

En lugar de definir las actividades del pipeline, esta herramienta se enfoca en definir los elementos de arquitectura de la solución a desplegar. En la herramienta, a cada elemento de la arquitectura le corresponde una *Unidad de Despliegue* (*Delivery Unit*) que encapsula todas las actividades y artefactos que permiten construir y desplegar ese componente. Cada unidad de despliegue, a su vez, tiene un conjunto de artefactos requeridos, artefactos intermedios y artefactos a proveer. A partir de esta especificación, la herramienta genera un modelo de actividades que define como se generan los artefactos intermedios y a proveer a partir de los artefactos requeridos.

En la especificación del funcionamiento interno de la herramienta, los diferentes artefactos que se pueden utilizar en los pipelines (el conjunto A) están clasificados en una serie de tipos de artefactos (el conjunto R). Por ejemplo, un artefacto puede ser una carpeta `source` de tipo *espacio de trabajo Java* o puede ser un archivo `app.jar` un *archivo ejecutable Java*. A su vez, una actividad de transformación de artefactos (en el conjunto T) permite producir un artefacto de un tipo particular a partir de uno o más artefactos de entrada. Por ejemplo, una actividad de transformación *compilar* puede tomar un *espacio de trabajo Java* y producir *un archivo ejecutable Java*. Finalmente, utilizando una base de conocimiento de actividades de transformación, la herramienta puede tomar un conjunto de artefactos de entrada y un conjunto de tipos de artefactos de salida y determinar cuáles son las transformaciones que permiten generar los artefactos deseados. Este proceso, donde se convierte una especificación del pipeline basada en artefactos en una especificación basada en actividades, puede ser usado para producir el código del pipeline.

[5] https://swc.rwth-aachen.de/docs/2021_APSEC_Aydin_Steffens.pdf

Debido al uso de modelos que representan tanto los artefactos como el proceso de construcción, la herramienta JARVIS no solo permite generar el código del pipeline, también permite operaciones de *backtracking* para encontrar diferentes opciones de procesamiento para producir un mismo artefacto, de creación y simplificación de grafos de dependencia entre los artefactos, y de optimización de los procesos de construcción.

Según la evaluación hecha por los autores, esta herramienta facilita la construcción de pipelines por parte de desarrolladores con poca experiencia. Los autores hicieron un estudio cualitativo industrial con un pequeño grupo de 8 desarrolladores. Mientras la mitad del grupo, desarrolladores Junior con menos experiencia en arquitecturas de software, lograron crear los artefactos necesarios sin problema, el resto del grupo con más experiencia encontró el que proceso no era tan intuitivo.

3.2.3.6. Otros trabajos relacionados

Por otra parte, en la búsqueda se encontraron otros trabajos relacionados que, aunque no tenían como objetivo crear un generador automático de pipelines, describen procesos de creación de pipelines desde diferentes enfoques. A continuación, se presentan algunos de esos artículos:

1. **Comparing DevOps procedures from the context of a systems engineer:** Priyadarsini et al. (2020), presenta dos arquitecturas Devops, una montada con Azure Devops y otra con AWS. Las cuales, se comparan usando los tiempos de construcción, de procesamientos y de fallas. Lo anterior, utilizando unos proyectos ya funcionando con Azure y/o AWS en la empresa Valamis. Sin embargo, el desarrollo del artículo se enfoca en la comparación entre las arquitecturas, que en la generación de diversos pipelines dependiendo la tecnología utilizada.

2. **Fast Delivery, Continuously Build, Testing and Deployment with DevOps Pipeline Techniques on Cloud:** por Jumani et al. (2020), presenta la implementación de pipelines para Devops en una organización, sobre la nube de Azure, centrándose en los requerimientos de un proyecto definido. Como resultado Jumani y sus compañeros describen la solución a grandes desafíos culturales y de infraestructura a los que se debieron enfrentar, teniendo en cuenta, que la solución para los problemas de infraestructura se centró en la infraestructura como código que provee la nube, lo cual genera una visión positiva al uso de infraestructuras como código para la construcción de estas soluciones.

3. **Implementation of a DevOps Pipeline for Serverless Applications:** por Ivanov and Smolander (2018), presenta la implementación de un pipeline automatizado para Devops, el cual incluye 27 prácticas para integración continua, entrega continua y monitoreo. Todo lo anterior mediante un enfoque serverless, lo cual generó que 18 de las 27 prácticas implementadas fueran influenciadas por las características específicas de serverless. Con lo que se reafirma las ventajas de utilizar infraestructura como código para dar solución a esta tarea.

4. **Building Lean Continuous Integration and Delivery Pipelines by Applying DevOps Principles: A Case Study at Varidesk:** por Debroy et al. (2018), presenta la implementación de devops en una de las aplicaciones web de la empresa Varidesk. En este

caso, Debroy y sus compañeros mencionan dos desafíos a los que se enfrentaron, siendo el primero los largos tiempos de espera para que las compilaciones/lanzamientos se pongan en cola y se completen. Y el segundo, la falta de soporte en algunas herramientas por parte de proveedores de nube. Estos temas los resolvieron con prácticas de contenedorización, infraestructura como código y orquestación. Lo cual, nuevamente brinda ciertas recomendaciones al tipo de infraestructura y posibles herramientas a seleccionar.

3.2.3.7. Herramientas relacionadas

Adicionalmente, se buscaron algunas herramientas que generen pipelines de CI/CD, fuera de la literatura formal. En ese proceso se encontraron 3 herramientas que generan pipelines. A continuación, se presentan estas herramientas.

1. **Jhipster:** Jhipster [6] es una plataforma de código abierto que permite a los desarrolladores crear rápidamente aplicaciones web y móviles modernas y escalables utilizando las mejores prácticas y tecnologías de la industria. Esta herramienta de desarrollo de aplicaciones, basada en Java y JavaScript, integra una amplia gama de tecnologías y frameworks populares, como Spring Boot, Angular, React y Vue.js, entre otros.

   Una de las características clave de JHipster es su capacidad para generar aplicaciones de manera eficiente, proporcionando a los desarrolladores un esquema de inicio rápido que incluye una arquitectura sólida, autenticación de usuarios, autorización, gestión de entidades, APIs RESTful y mucho más. Además, JHipster facilita la integración con bases de datos relacionales y NoSQL, así como con servicios en la nube como AWS y Azure.

   La arquitectura de JHipster incluye el patrón de microservicios, lo que permite a los desarrolladores crear aplicaciones escalables y distribuidas, al igual que monolitos, el cual da un punto central de gestión del proyecto. Además, JHipster ofrece características de seguridad avanzadas, como la gestión de tokens JWT o Oauth2 y la protección contra ataques comunes, garantizando la seguridad de las aplicaciones desarrolladas con esta plataforma.

   En resumen, JHipster es una herramienta poderosa y versátil para el desarrollo rápido de aplicaciones web y móviles, que permite a los desarrolladores concentrarse en la lógica de negocio de sus aplicaciones, mientras que JHipster se encarga de la configuración y la integración de las tecnologías subyacentes.

   Adicionalmente, el Generador de Pipelines integrado en JHipster es una característica que permite a los desarrolladores automatizar el proceso de integración continua y entrega continua (CI/CD) de sus aplicaciones desarrolladas con JHipster. Este generador facilita la configuración de pipelines de CI/CD utilizando herramientas populares como Jenkins, GitLab CI, CircleCI o GitHub Actions.

   Al utilizar el Generador de Pipelines, los desarrolladores pueden definir fácilmente los pasos necesarios para la construcción, prueba y despliegue de sus aplicaciones, todo ello dentro

[6] https://www.jhipster.tech/

del flujo de trabajo de CI/CD. Esto incluye la compilación del código fuente, la ejecución de pruebas automatizadas, la generación de artefactos de construcción y su despliegue en entornos de desarrollo, pruebas o producción.

Una de las ventajas del Generador de Pipelines es su integración perfecta con las aplicaciones generadas por JHipster, lo que permite una configuración rápida y sencilla de los pipelines de CI/CD. Además, el Generador de Pipelines proporciona plantillas predefinidas que cubren los escenarios más comunes de desarrollo de aplicaciones, lo que facilita aún más la configuración inicial de los pipelines.

En resumen, el Generador de Pipelines integrado en JHipster es una herramienta poderosa que simplifica y automatiza el proceso de CI/CD para las aplicaciones desarrolladas con esta plataforma, permitiendo a los equipos de desarrollo mejorar la eficiencia, la calidad y la velocidad de entrega de sus aplicaciones.

2. **Scaffold de Bancolombia:** El Scaffold de Bancolombia [7] es una herramienta de código abierto basada en Gradle que facilita y acelera el desarrollo de software utilizando la arquitectura limpia ("Clean Architecture"). Proporciona una estructura de directorios y archivos predefinida, automatiza la creación de código base y simplifica la implementación de pruebas unitarias. El utilizar esta herramienta trae los siguientes beneficios:

   - Simplifica la implementación de la arquitectura limpia: Automatiza la creación de la estructura de directorios y archivos necesarios.
   - Promueve la modularidad: Fomenta la organización del código en módulos independientes y reutilizables.
   - Facilita las pruebas unitarias: Simplifica la escritura y ejecución de pruebas unitarias para el código.
   - Mejora la mantenibilidad: Hace que el código sea más fácil de entender, modificar y mantener a largo plazo.
   - Aumenta la productividad: Permite a los desarrolladores centrarse en la lógica de negocio y no en la estructura del código.

   El Scaffold de Bancolombia define una estructura de directorios estándar para una arquitectura limpia:

   - Entities: Contiene las clases que representan el dominio del problema.
   - UseCases: Contiene las clases que implementan la lógica de negocio.
   - Adapters: Contiene las clases que traducen las entidades y casos de uso a formatos compatibles con el mundo exterior.
   - Frameworks: Contiene las dependencias externas que no interfieren en el núcleo de la aplicación.

[7] https://github.com/bancolombia/scaffold-clean-architecture

- Tests: Contiene los scripts de prueba unitarios para las entidades y casos de uso.

Adicionalmente tiene otros generadores, entre ellos un generador de pipelines, el cual, permite crear pipelines de integración y entrega continua (CI/CD) utilizando herramientas como Jenkins, GitLab CI, Azure pipelines y CircleCi. El generador automatiza la construcción, las pruebas y la implementación de código.

3. **Jenkins Pipelines Generator de Octopus:** El Jenkins Pipelines Generator de Octopus [8] es una herramienta gratuita que facilita la generación de pipelines de Jenkins personalizados para la implementación de proyectos de software en Octopus Deploy. Esta herramienta te permite ahorrar tiempo, evitar errores y mejorar la consistencia y mantenibilidad de tus pipelines. Los beneficios de esta herramienta son:

   - Ahorro de tiempo: Automatiza la creación de pipelines de Jenkins, lo que te permite dedicar más tiempo al desarrollo de software.
   - Reducción de errores: Genera pipelines de Jenkins que se integran a la perfección con Octopus Deploy, minimizando el riesgo de errores durante la implementación.
   - Mayor consistencia: Garantiza que tus pipelines de Jenkins sean consistentes y fáciles de mantener, simplificando la gestión de tus procesos de implementación.

   Funcionamiento:

   El Jenkins Pipelines Generator de Octopus funciona escaneando el repositorio de código y generando un pipeline de Jenkins personalizado en función de la configuración del proyecto. El pipeline generado incluye:

   - Pasos de construcción: Preparan tu código para la implementación, como la compilación y la ejecución de pruebas unitarias.
   - Pasos de implementación: Despliegan tu código en Octopus Deploy, siguiendo los pasos definidos en tu proyecto.
   - Pasos de prueba: Ejecutan pruebas automatizadas para verificar el correcto funcionamiento de tu código después de la implementación.
   - Pasos de notificación: Informan a las partes interesadas sobre el estado de la implementación, como el éxito o el fracaso.

3.3. P02: ¿Qué cobertura tienen estos generadores y como fueron creados?

3.3.1. Extracción y Síntesis de Datos

Para la obtención de la información se desarrolló un banco de datos, dentro del cual se ajustó la información relevante extraída de cada uno de los artículos incluidos en la revisión. Las característi-

[8] https://jenkinspipelinegenerator.octopus.com

cas de interés para esta revisión fueron: Las herramientas de CI/CD utilizadas, los entornos en que se despliega la aplicación, los tipos de empaquetado de la aplicación que usan, donde almacenan los artefactos de la aplicación, la metodología usada para crear los pipelines, qué lenguaje usan en el desarrollo del generador y si automatizan la generación y cómo lo hacen.

En este paso se analizó la información suministrada en la documentación de las diferentes herramientas, enfocándose en las características más claras de las herramientas. En la tabla 3.2 se puede revisar esta información.

	Artículo 1 (Donca et al., 2022)	**Artículo 2** (Djounang-Nana, 2022)	**Artículo 3** (Laaja, 2022)	**Artículo 4** (Jones, 2018)	**Artículo 5** (Aydin et al., 2021)	**Artículo 6** (Padhye et al., 2023)
Herramientas CI/CD	Gitlab CI/CD	Circle CI, Github Actions y Gitlab CI/CD	Gitlab CI/CD	Jenkins		Jenkins
Entorno de despliegue	Kubernetes		Apple Store y Google Play			
Empaquetado	Docker		ipa y apk			
Almacenamiento artefactos	Elastic Container Registry		local			
Formas de creación	Plantillas	Meta-heurística, simbólica y estadística		Plantillas y lenguajes específicos de dominio	Arquitectura de software y vista centrada en artefactos	
Lenguaje del generador	Bash Scripting language		Javascript (Ionic)			
Automatización de generación	Analiza archivos de construcción y despliegue y genera los jobs desde el template	Analiza el proyecto, selecciona jobs y traduce a la herramienta de CI/CD		Archivo de definición y scripts personalizados		Configuración declarativa

Tabla 3.2: Comparación de artículos

En la tabla 3.2 se presenta la síntesis de las herramientas relacionadas.

	Jhipster	**Scaffold Bancolombia**	**Jenkins Pipelines Generator**
Herramientas CI/CD	Azure pipelines, Github Actions, Gitlab CI/CD, Circle CI, Travis CI, Jenkins	Jenkins, GitLab CI, Azure pipelines y CircleCi	Jenkins
Entorno de despliegue	Heroku		Octopus Deploy
Empaquetado	Docker y Jar	Docker	Jar o War
Almacenamiento artefactos	Artefactory		
Formas de creación	Selecciona archivo y código dependiendo las opciones del usuario	Plantillas	Plantillas y entrada de usuario
Lenguaje del generador	Javascript	Java	Javascript y Java

Tabla 3.3: Comparación de herramientas relacionadas

3.4. P03: ¿Qué limitaciones tienen los generadores disponibles?

Al analizar los diferentes generadores de pipelines, los descritos en los artículos y los generadores relacionados, se presentan algunas limitaciones que estos tienen basados en los siguientes criterios, los cuales son importantes en el presente trabajo de grado:

- **Criterio 1: Usa tres o más herramientas de CI/CD**

  Justificación: La importancia de este criterio se debe a que la idea del generador es que permita generar pipelines para diferentes herramientas de CI/CD

- **Criterio 2: Genera pipeline hasta despliegue de la aplicación**

  Justificación: Este criterio es importante porque la idea del generador es que lleve una aplicación desde el código hasta desplegarlo en la nube.

- **Criterio 3: Despliega en 3 o más servicios de nube**

  Justificación: La importancia de este criterio es que el generador permita seleccionar diferentes tipos de servicios en la nube para desplegar la aplicación.

- **Criterio 4: Está disponible para entornos laborales**

  Justificación: Este criterio se de a que se espera que la herramienta sea accesible por todas las personas que la requieran y no solo en entornos académicos

- **Criterio 5: Empaqueta la aplicación en dos o más tipos (por ejemplo Docker y Jar)**

  Justificación: La idea es que la herramienta permita variabilidad en más campos, para así ampliar cobertura de acción.

- **Criterio 6: Es fácil de obtener y usar**

  Justificación: La importancia en este caso es porque se espera que la herramienta sea pública y se pueda usar y/u obtener sin complicaciones, teniendo en cuenta que la herramienta es para personas técnicas.

En la tabla 3.4 se correlacionan los artículos y herramientas relacionadas con los diferentes criterios descritos previamente, con el objetivo de percibir las limitaciones que estos tenían:

	Criterio 1	Criterio 2	Criterio 3	Criterio 4	Criterio 5	Criterio 6
Artículo 1 (Donca et al., 2022)	□	■	□	□	□	□
Artículo 2 (Djounang-Nana, 2022)	■	⊠	□	□	⊠	□
Artículo 3 (Laaja, 2022)	□	■	□	≈	■	□
Artículo 4 (Jones, 2018)	□	⊠	□	≈	⊠	□
Artículo 5 (Aydin et al., 2021)	□	⊠	□	□	⊠	□
Jhipster	■	■	□	■	■	■
Scaffold Bancolombia	■	□	□	■	□	■
Jenkins Pipelines Generator	□	■	□	■	■	■

■ Cumple totalmente ≈ Cumple parcialmente ⊠ No especifica □ No cumple

Tabla 3.4: Comparación de artículos basado en los criterios

Analizando la tabla 3.4, se logró percibir que gran parte de las herramientas encontradas no cumplían todos los criterios definidos, confirmando de esa manera la posibilidad de generar una herramienta que permita solventar estas limitaciones encontradas.

Con respecto al primer criterio se percibe que solo 3 de los generadores cumplen completamente y el resto no cumplen, lo cual indica que, según la descripción del generador, ellos no usan varias herramientas de CI/CD y enfocan su desarrollo en solo una. Esto limita el uso del generador, ya que, si no se desea usar la herramienta seleccionada por el generador, no es posible usarlo.

El segundo criterio, al igual que el quinto tienen una característica especial y es que algunos de los generadores encontrados no especifican si el pipeline que generan tiene pasos para desplegar la aplicación y en qué forma empaquetan la aplicación y otros no lo tienen. Sin embargo, estas

características son bastante importantes, ya que se espera que con el pipeline se puedan empaquetar las aplicaciones de diferentes formas y que se despliegue la aplicación en algún servicio de nube disponible.

El tercer criterio es la limitación más visible en los generadores encontrados ya que, aunque algunos tienen disponible la posibilidad de desplegar la aplicación, solo lo hacen en un servicio, lo cual pierde el enfoque de brindar varias opciones de despliegue.

El cuarto criterio tiene dos artículos, los cuales lo cumplen parcialmente debido a que, aunque los generadores se usan entornos laborales, solo se usan para la empresa que fue creada. Adicionalmente, en los otros artículos no se describe que estén disponibles para uso en este ámbito. Por estas mismas razones tampoco se cumple el criterio seis, ya que los artículos no describen un entorno público para obtener los generadores, el proceso sería complicado, ya que se debería hacer una solicitud a través de la universidad o dueños del generador.

3.5. Resumen del capítulo

En la primera sección de este capítulo se presentó las bases teóricas que permitan comprender los temas relacionados al presente trabajo de grado, como lo son la Integración, Entrega y Despliegue Continuo, al igual que los pipelines y algunos métodos de reutilización y variabilidad de software.

En la segunda sección se plantearon las preguntas orientadoras y se realizó un estado del arte sobre generadores de pipelines para CI/CD, el cual ayudó a responder la primera pregunta y posteriormente un análisis en ellos sobre sus características y sus limitaciones, para así responder la segunda y tercer pregunta. Con lo anterior se percibió las posibilidades de mejora y crecimiento con el presente trabajo de grado.

Capítulo 4

Desarrollo del Proyecto

Considerando los retos que enfrentan los desarrolladores al momento de crear pipelines de CI/CD, este proyecto se plantea la creación de una herramienta que permita generar, a partir de las especificaciones del desarrollado, un pipeline que permita el despliegue en nube.

Este capítulo describe las definiciones, los requerimientos y el proceso de diseño y desarrollo realizado para construir la herramienta.

4.1. Definiciones para el desarrollo del generador de pipelines

Como primer paso del proceso de desarrollo, se definieron los requerimientos y herramientas a soportar por la herramienta a construir.

4.1.1. Herramientas base para el desarrollo

Para el desarrollo de la herramienta se decidió basar la funcionalidad en algún generador ya existente. En la actualidad, existen varios generadores de aplicaciones Java con Spring Boot. Se pueden mencionar, por ejemplo, Spring Initializr[1], Bootify[2] y Yeoman[3] En nuestro caso, optamos por usar JHipster[4], una solución ampliamente utilizada a nivel mundial.

JHipster (actualmente versión 7.9.3) es una plataforma que permite generar, desarrollar y desplegar aplicaciones web modernas. En esta herramienta permite generar la solución con una interfaz de usuario (frontend) usando tecnologías como Angular, React o Vue, y servicios de aplicación (backend) en Spring Boot, usando como herramienta de compilación, o maven o gradle. Adicionalmente, JHipser permite el despligue de la aplicación usando docker o kubernetes y se puede desplegar en nubes como la de AWS, GCP, Azure o Heroku, entre otros. Para integración y entrega continuo, JHipster se puede usar con herramientas como Jenkins, Travis CI, Gitlab CI, Circle CI, Azure Pipelines y Github Actions, apoyados de herramientas para pruebas unitarias, de integración, de rendimiento y más, con herramientas como JUnit, Jest, Cucumber, Spring Test Context framework, Cypress, entre otros[5].

A continuación, se presenta un modelo de características que representa las diferentes opciones y alternativas que se pueden emplear en JHipster (jhipster-generator) para crear una aplicación.

[1] https://start.spring.io/
[2] https://bootify.io/
[3] https://yeoman.io/
[4] https://www.jhipster.tech/
[5] https://www.jhipster.tech/

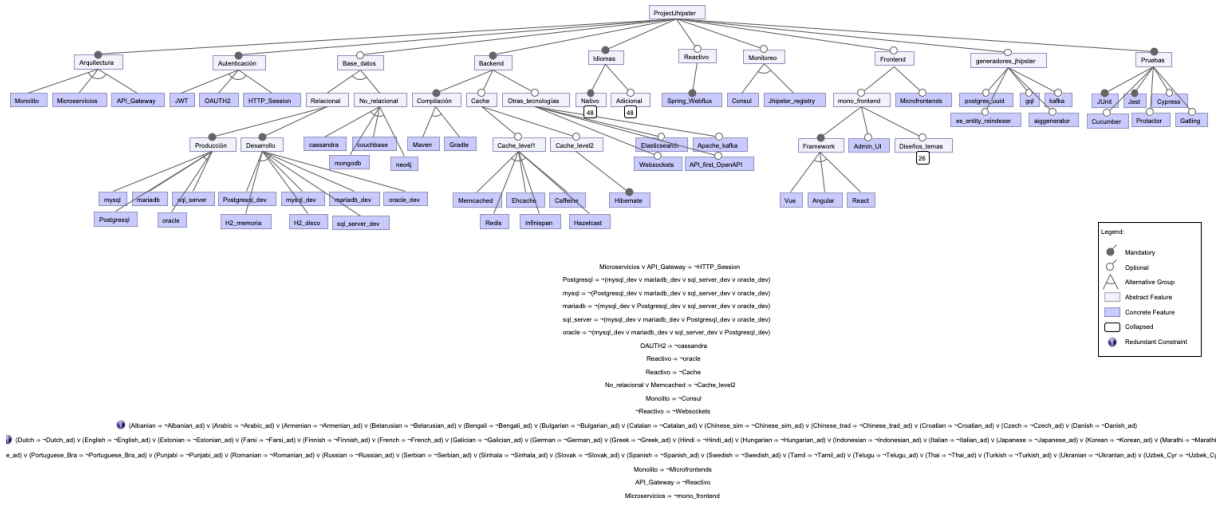


Figura 4.1: Modelo de Características de JHipster generator. Fuente propia

Para desarrollar y probar nuestra herramienta, se elaboraron dos *proyectos base* que se utilizó para generar las diferentes opciones despliegue. Los dos proyectos base se generaron usando frontend Angular y backend en Spring Boot. Uno de ellos se construye usando Maven y el otro usando Gradle. Cada uno de estos proyectos está nombrado de la siguiente manera:

- ProyBaseAngMvn: Proyecto base con Angular y Maven
- ProyBaseAngGrl: Proyecto base con Angular y Gradle

Además, teniendo en cuenta las diferentes opciones con las que dispone JHipster en su generador, los dos proyectos base fueron generados con las siguientes características:

- Arquitectura monolítica
- Funcionamiento sin el uso de Spring Webflux
- Autenticación con JWT
- Base de datos Postgresql
- Caché con Ehcache
- Segundo nivel de cache con Hibernate Cache
- Incluyendo interfaz de usuario de administración
- Pruebas usando Junit, Jest, Cypress, Gatling y Cucumber

4.1.2. Herramientas de CI/CD soportadas

Además del código fuente, JHipster permite generar algunos archivos y pipelines. La Figura 4.2 muestra las opciones disponibles:

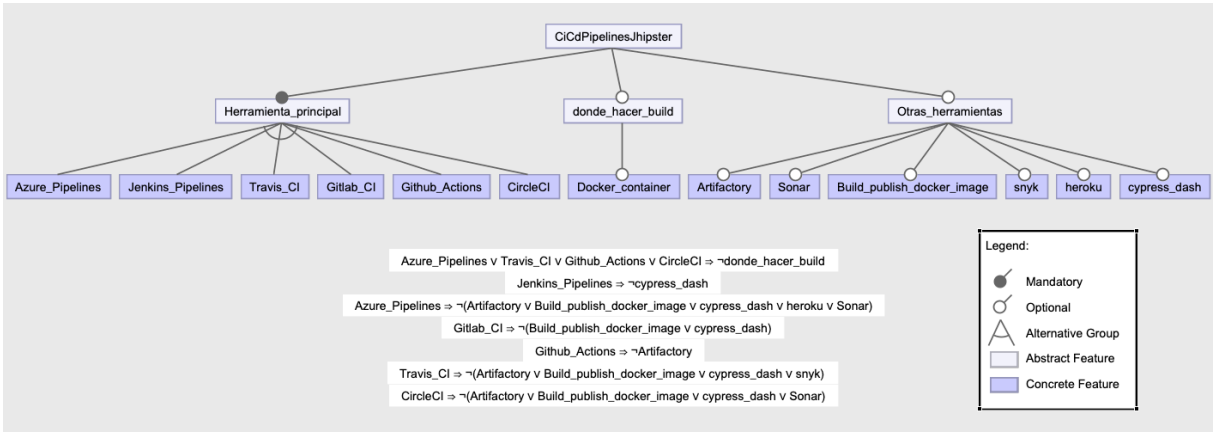


Figura 4.2: Modelo de características del generador de pipelines de Jhipster. Fuente propia

Al revisar los pipelines generados por JHipster con los proyectos base mencionados arriba pudimos observar que estos pipelines permiten ejecutar pruebas automatizadas y generar los paquetes ejecutables, pero no el despliegue de la aplicación en nube. JHipster permite generar pipelines de integración continua, pero no pipelines CI/CD que incluyan el despliegue de la aplicación.

Cabe anotar que la documentación de JHipster describen múltiples opciones para desplegar las aplicaciones generadas. Sin embargo, estos procesos de despliegue son independientes y no están integrados a pipelines que se pueden generar con la herramienta. Debido a lo anterior, se percibe la posibilidad de ampliar el funcionamiento de los pipelines generados.

Como parte del trabajo de grado se decide extender JHipster para generar pipelines que permitan desplegar las aplicaciones en nube, usando las herramientas de CI/CD que ya estaban soportadas. Específicamente, se estableció extender JHipster para generar pipelines para Jenkins, Azure Pipelines, Github Actions, CircleCI y TravisCI. Aunque JHipster soporta pipelines Gitlab-CI se optó por no generar pipelines para esta herramienta debido a que el periodo de prueba de un mes que ofrece Gitlab no era suficiente para hacer todo el desarrollo y las pruebas correspondientes.

4.1.3. Servicios de Nube soportados

En la actualidad existen unos cuantos proveedores de nube líderes del mercado. Tal como puede verse en la Figura 4.3, de acuerdo con el informe del "Cuadrante Mágico de Gartner", las plataformas de nube líderes en 2023 fueron las ofrecidas por AWS, Microsoft, Google y Oracle[6].

[6] https://aws.amazon.com/es/blogs/aws/read-the-2023-gartner-magic-quadrant-for-strategic-cloud-platform-services/

Figura 4.3: Cuadrante mágico de Gartner, octubre 2023. Fuente, reporte de Gartner 2023

Al revisar las diferentes opciones de despliegue de las aplicaciones web en estos proveedores se pudo observar la gran cantidad de opciones que ofrece cada una. La misma aplicación puede desplegarse usando servicios de máquinas virtuales (IaaS), de plataforma como servicio (PaaS), de contenedores, de clústeres de contenedores, entre otros. Con el fin de reducir la variabilidad requerida para la herramienta se optó por trabajar con la nube de Azure y las diversas opciones que tiene para el despliegue de aplicaciones Java como las que puede generar JHipster.

Azure ofrece los siguientes servicios con los cuales se puede desplegar una aplicación web basada en Java:

- **Azure Kubernetes Service (AKS)** [7]**:** AKS es un servicio administrado de Kubernetes[8] que permite implementar, administrar y escalar contenedores en Azure. Simplifica la gestión

[7] https://learn.microsoft.com/en-us/azure/aks/
[8] https://kubernetes.io/es/docs/home/

de clústeres de Kubernetes, elimina la necesidad de administrar la infraestructura subyacente y permite centrarse en la creación y ejecución de aplicaciones.

Algunos casos de uso de AKS son:

- Implementar y administrar aplicaciones nativas de la nube escalables.
- Modernizar aplicaciones monolíticas existentes en contenedores.
- Crear y ejecutar microservicios.
- Administrar entornos de contenedores complejos.

- **Azure App Service WebApp[9]:** WebApp es un servicio de plataforma como servicio (PaaS[10]) para implementar y ejecutar aplicaciones web y móviles en la nube. Soporta diversas plataformas de desarrollo, incluyendo .NET, Java, Python, Node.js, PHP y Ruby.

  Algunos casos de uso de WebApp son:

  - Implementar y ejecutar aplicaciones web y móviles rápidamente.
  - Escalar automáticamente las aplicaciones en función de la demanda.
  - Integrar con otros servicios de Azure, como Azure Storage, Azure Cosmos DB y Azure Functions.
  - Implementar aplicaciones sin necesidad de administrar infraestructura.

- **Azure App Service WebApp para Contenedores[11]:** WebApp para Contenedores se encuentra dentro del mismo servicio de App Service, pero cambia su configuración en el momento de crearlo, permitiendo implementar y ejecutar aplicaciones en contenedores en Azure App Service. Ofrece la flexibilidad de los contenedores con la simplicidad y escalabilidad de Azure App Service.

  Algunos casos de uso de WebApp para contenedores son:

  - Implementar y ejecutar aplicaciones en contenedores en un entorno de PaaS.
  - Aprovechar las ventajas de los contenedores, como la portabilidad y el aislamiento.
  - Escalar automáticamente las aplicaciones en función de la demanda.
  - Integrar con otros servicios de Azure, como Azure Storage, Azure Cosmos DB y Azure Functions.

Teniendo la información anterior, se decide incluir los tres servicios de Azure en el generador para brindar a los usuarios la posibilidad de usarlos y desplegar la aplicación en cualquiera de ellos.

[9] https://learn.microsoft.com/en-us/azure/app-service/
[10] https://azure.microsoft.com/es-es/resources/cloud-computing-dictionary/what-is-paas
[11] https://azure.microsoft.com/en-us/products/app-service/containers

4.1.4. Empaquetado de aplicaciones Java

Entendiendo que la entrega del software es un paso importante del pipeline, se revisaron las diferentes opciones para el empaquetado y distribución de la aplicación. Para empaquetar aplicaciones Java se puede hacer en contenedores, así como en archivos JAR[12], WAR[13] y EAR[14]. En nuestro caso, se utilizan Docker y JAR, ya que con ellos se cubre el uso de los servicios de Azure mencionados en la subsección 2.3.

- **Docker**[15] es una plataforma de contenedores que permite empaquetar una aplicación y sus dependencias en un contenedor ligero, autosuficiente y ejecutable llamado imagen de Docker. Las imágenes de Docker se pueden ejecutar en cualquier entorno que tenga instalado Docker, lo que garantiza que la aplicación se ejecute de manera consistente independientemente del sistema operativo o la configuración del entorno subyacente. En el presente proyecto se usará Docker para empaquetar la aplicación, subir la imagen al Azure Container Registry y desplegarlo con AppService WebApp para contenedores o con Azure Kubernetes Service.

- **JAR**[16] (Java Archive) es un formato de archivo estándar para empaquetar clases y recursos de Java en un solo archivo. Los archivos JAR se utilizan comúnmente para distribuir bibliotecas de Java, aplicaciones web Java y aplicaciones Java de escritorio. En nuestro caso se usará para empaquetar la aplicación Java, entregarlo como artefacto y así desplegarlo en el App Service WebApp.

4.1.5. Herramientas de compilación y gestión de proyectos

Además, se definió soportar Maven y Gradle para compilar y gestionar los paquetes y dependencias del software.

- **Gradle**[17] es una herramienta de automatización de construcción basada en Apache Ant y Groovy. Se caracteriza por su sintaxis declarativa y flexible, que permite definir las tareas de construcción de manera concisa y legible. Gradle también ofrece soporte para la gestión de dependencias, la ejecución de pruebas unitarias y la generación de informes.

- **Maven**[18] es una herramienta de automatización de construcción basada en XML y Java. Se caracteriza por su enfoque convencional y estandarizado para la gestión de proyectos y la construcción de software. Maven utiliza un conjunto de plugins predefinidos para realizar tareas comunes como la compilación, la ejecución de pruebas y la creación de paquetes.

[12] https://www.ibm.com/docs/es/i/7.5?topic=platform-java-jar-class-files
[13] https://www.ibm.com/docs/es/rsas/7.5.0?topic=projects-web-archive-war-files
[14] https://www.ibm.com/docs/es/baw/20.x?topic=modules-ear-file-overview
[15] https://www.docker.com/
[16] https://docs.oracle.com/javase/tutorial/deployment/jar/basicsindex.html
[17] https://gradle.org/
[18] https://maven.apache.org/

4.1.6. Técnica para reutilización y variabilidad

En la definición de la técnica adecuada para la gestión de la reutilización y la variabilidad en el software, en el marco teórico se puede observar que existen varias técnicas y tipos de reutilización que se pueden usar, al igual que formas para gestionar la variabilidad que tiene el software, obteniendo grandes beneficios de esta manera. Sin embargo, ya que la idea es relacionar y sacar provecho de estas dos estrategias, se opta por la técnica de líneas de productos de software, la cual, intercepta las estrategias de reutilización y variabilidad.

4.1.6.1. Línea de Productos de Software (LPS)

Esta técnica se enfoca en definir las características comunes y variables en un grupo de productos de software y de esa manera construir una arquitectura común y componentes compartidos, que se pueda especializar a cada una de las aplicaciones y requerimientos del cliente. Esto puede implicar un gasto de tiempo y esfuerzo en los equipos de desarrollo en un inicio, pero que es recompensado luego, ya que, al tener componentes comunes el mantenimiento y mejoras de estos se hace menos costoso al realizarse una sola vez y no en cada uno de los productos.

Por lo general, según Sommerville (2016), una aplicación creada con LPS contiene:

- **Componentes Core:** Estos brindan soporte de infraestructura y por lo general no son modificados al crear una nueva instancia de la línea de productos.
- **Componentes configurables:** Estos componentes se pueden modificar o configurar para enfocarlos hacia una aplicación. A veces, es posible reconfigurar estos componentes sin cambiar su código usando un lenguaje de configuración de componentes integrado.
- **Componentes especializados:** Algunos o todos estos componentes pueden ser reemplazados cuando se crea una nueva instancia de la línea de productos.

Por otra parte, según Apel et al. (2013), las características específicas de las LPS se pueden dividir en ingeniería de dominio e ingeniería de aplicación o en espacio del problema y espacio de la solución, lo cual, es posible mapear en una estructura de dos dimensiones como se observa en la figura 4.4.

A continuación, se describen las dimensiones presentes en la figura 4.4

- **Ingeniería de Dominio:** Actividades tendientes a construir los activos (los elementos reutilizables en los diferentes productos).
- **Ingeniería de Aplicación:** Actividades tendientes a construir un producto a partir de los activos construidos en la Ingeniería de Dominio.
- **Espacio del Problema:** Actividades relacionadas con el domino del problema, los requerimientos y expectativas de los usuarios.
- **Espacio de la Solución:** Actividades relacionadas con el domino de la solución, el diseño y la implementación del software.

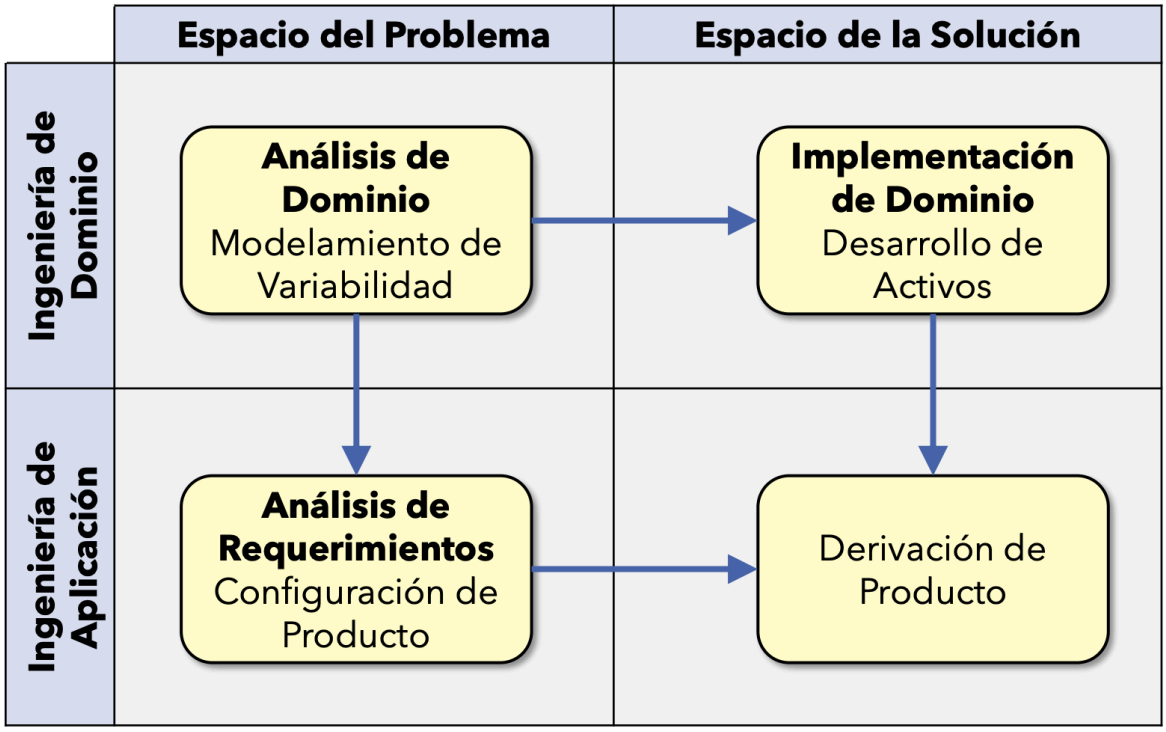


Figura 4.4: Procesos en LPS. Imagen adaptada de (Apel et al., 2013)

Adicionalmente, se describe cada uno de los campos en el diagrama de procesos de las LPS de la figura 4.4:

- **Análisis de Dominio:** Determinar la variabilidad, o sea las características comunes y variables de los productos.
- **Análisis de Requerimientos:** Determinar los requerimientos y las características que deben ir en un producto particular.
- **Implementación de Dominio:** Implementar la variabilidad identificada (las características comunes y variables).
- **Derivación de producto:** Desarrollar el producto utilizando los activos identificados para ese producto particular.

En el caso de la implementación de dominio, Apel et al. (2013) describen algunos conceptos importantes en el proceso de implementación de la variabilidad, siendo estos las dimensiones de implementación de la variabilidad que ayudan a describir como se va a gestionar esta y en que fases del desarrollo van a variar las características.

A continuación, se describen estas dimensiones:

- **Tiempo de Enlace (Binding Time):**
  - **Tiempo de compilación (variabilidad estática):** El desarrollador programa qué características se incluirán y esto sucede durante o antes del tiempo de compilación. En este caso las características que no se seleccionen, ni siquiera se compilan.
  - **Tiempo de carga:** Para el tiempo de carga, todas las características variables se compilan y ya cuando se despliega el software y empieza a cargar, se decide qué características presentar.
  - **Tiempo de ejecución (variabilidad dinámica):** Los cambios se realizan en tiempo de ejecución, por tanto, mientras se está ejecutando el programa se puede presentar la variabilidad.
- **Tecnología:**
  - **Basado en el lenguaje de programación:** Usa los mecanismos que brinda un lenguaje de programación para implementar características y generar productos.
  - **Basado en la herramienta:** Usa una o más herramientas externas para implementar o representar características en el código y controlar el proceso de generación de productos.
- **Representación:**
  - **Anotación:** El desarrollador programa que características se incluirán y esto sucede durante o antes del tiempo de compilación. En este caso las características que no se seleccionen, ni siquiera se compilan.
  - **Composición:** Para el tiempo de carga, todas las características variables se compilan y ya cuando se despliega el software y empieza a cargar, se decide qué características presentar.

Teniendo en cuenta estos procesos descritos en las LPS, en la sección 4.2.1, se realiza el Análisis del dominio para la implementación del generador de pipelines. En la sección 4.2.2, se hace el análisis de requerimientos y se determinan las diferentes características que debe contener cada uno de los pipelines a generar. En la sección 4.2.3, se crean los diferentes activos con los cuales se van a generar los pipelines. Y por último, en la sección 4.2.4, se genera cada uno de los pipelines definidos.

4.2. Creación de Línea de Productos de Software

4.2.1. Análisis de Dominio

Teniendo como base las herramientas descritas en la sección 4.1, se determina la variabilidad con la cual se trabajará en el presente trabajo de grado, adicionalmente, de las características comunes también presentes. En la figura 4.5 se presenta el modelo de características del generador

de pipelines a desarrollar, teniendo en cuenta que se toma como base los pipelines que ya genera Jhipster.

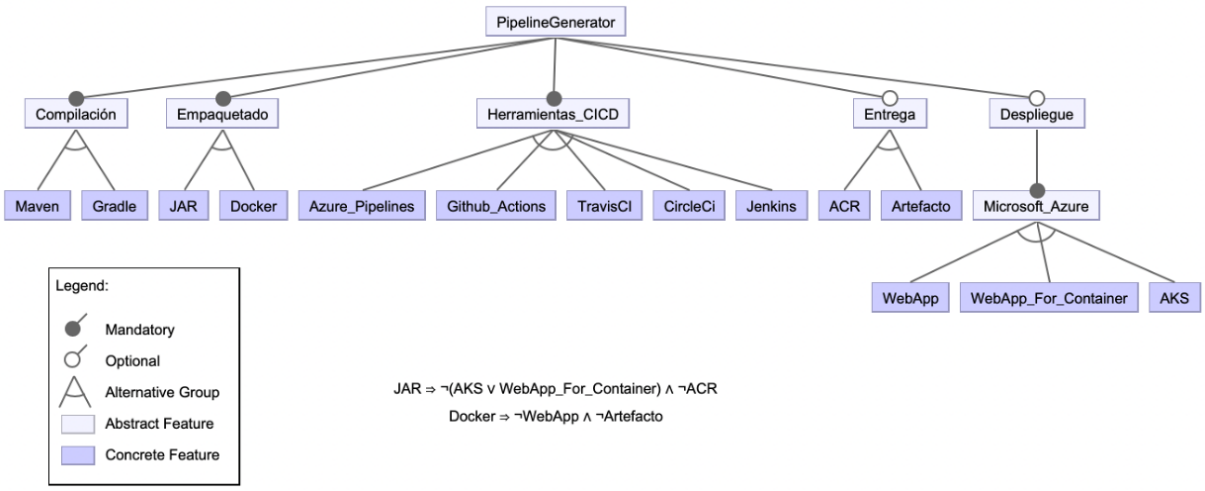


Figura 4.5: Modelo de características del generador de pipelines.

Debido a que los pipelines hasta la sección de pruebas ya los tiene configurados la herramienta de Jhipster, el enfoque es lograr la entrega y despliegue continuo, por lo tanto, la variabilidad se enfoca en las herramientas de despligue y los sistemas en los cuales se va a desplegar la aplicación. Así, para desplegar los archivos JAR, se entrega un artefacto que se gestiona dentro de la herramienta de despliegue y que luego será desplegado en WebApp; y para los casos de Docker, se genera un contenedor que se carga en Azure Container Registry y finalmente se despliega en WebApp For Container o en Azure Kubernetes Service. Basados en el anterior modelo y restricciones en él, se generan 70 diferentes configuraciones a ser desarrolladas.

4.2.2. Análisis de Requerimientos

En esta sección se busca analizar los requerimientos que tienen los usuarios con el generador, definiendo qué características son necesarias para aumentar la utilidad y por ende la satisfacción en ellos. Para obtener esta información, se utilizó la información del Capítulo 2 y las secciones 4.1 y 4.2.1.

A continuación, se presentan los diferentes requisitos identificados:

- **Requisito 1:** Organizar el pipeline en jobs o stages que permitan identificar la fase de pruebas, construcción, entrega y/o despliegue.
- **Requisito 2:** Permitir el definir cual rama del repositorio se desea escanear con el pipeline a generar.

- **Requisito 3:** Ejecutar el pipeline cada que se haga un push o un pull request a la rama.
- **Requisito 4:** Incluir las pruebas unitarias y de integración que el generador de Jhipster tenga, haciendo las correcciones pertinentes por si alguno falla.
- **Requisito 5:** Dar la opción de seleccionar las siguientes herramientas de CI/CD, Azure pipelines, Github Actions, Circle CI, TravisCI y Jenkins.
- **Requisito 6:** Permitir generar pipelines para dos tipos de empaquetado, Docker y Jar.
- **Requisito 7:** Brindar la opción de generar pipelines para proyectos creados con Gradle y Maven.
- **Requisito 8:** Para el despliegue de la aplicación, el generador debe permitir seleccionar Azure Kubernetes Service, Azure AppService WebApp y Azure AppService WebApp para contenedores.
- **Requisito 9:** Si se selecciona Azure App Service WebApp, por defecto se debe empaquetar en un archivo Jar.
- **Requisito 10:** Si se selecciona Azure Kubernetes Service o Azure AppService WebApp para contenedores, por defecto se debe empaquetar en un archivo docker.
- **Requisito 11:** Al empaquetar con Docker, se debe cargar la imagen en Azure Container Registry.
- **Requisito 12:** Al usar Azure Kubernetes Service, se deben generar los archivos necesarios de manifiesto para el correcto despligue y funcionamiento, como la configuración del balanceador de carga.
- **Requisito 13:** En el caso de Jenkins, se debe generar el pipeline con la sintaxis de tipo declarativo.

4.2.3. Implementación de Dominio

Para implementar las diferentes características comunes y variables del proyecto, se tomó como base las diferentes configuraciones que tiene Jhipster para cada uno de los pipelines, los cuales son modificados para lograr generar los pipelines para cada una de las herramientas seleccionadas.

4.2.3.1. Estructura del generador

El generador de ci-cd de Jhipster en su versión 7.9.3 tiene la siguiente estructura:

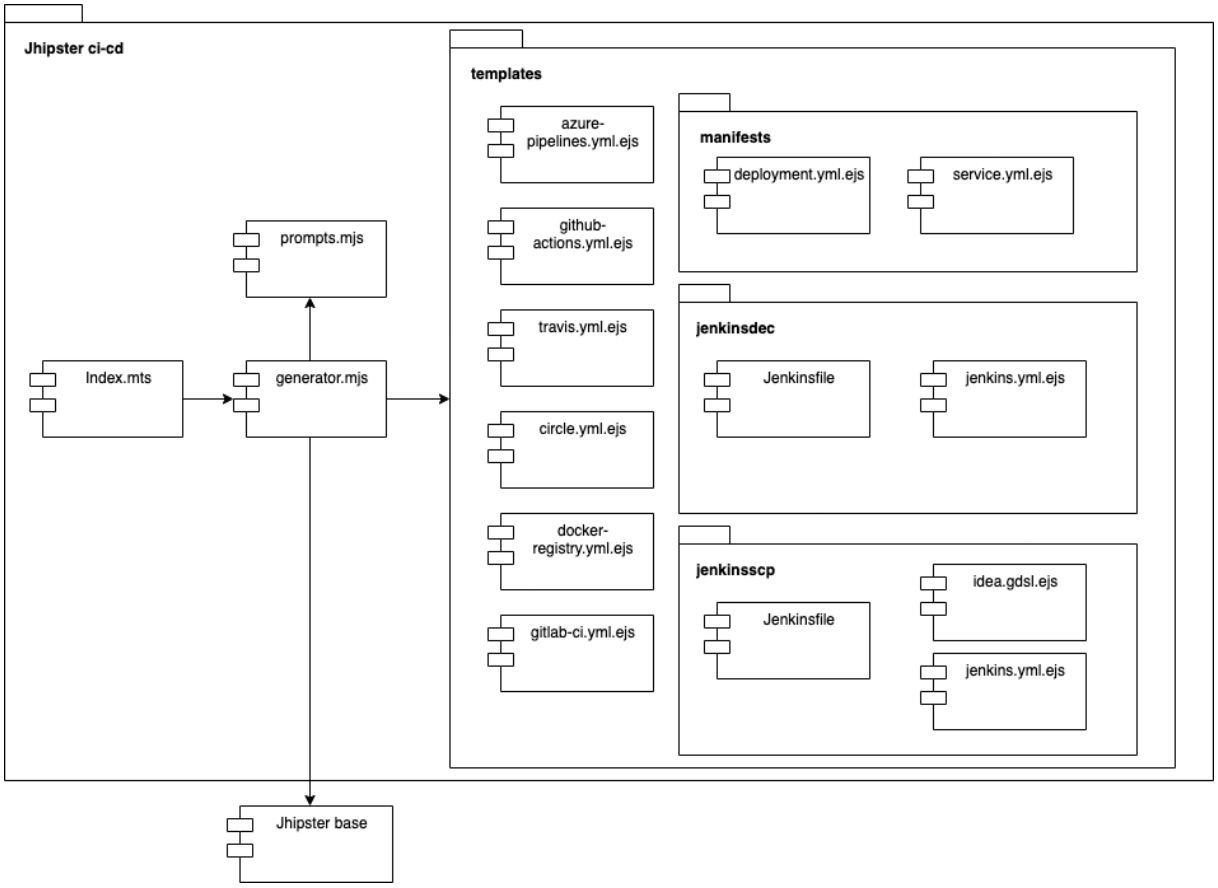


Figura 4.6: Diagrama de componentes Jhipster ci-cd. Fuente propia

A continuación, se describen las diferentes carpetas, componentes y archivos que se visualizan en la imagen 4.6:

- **Jhipster base:** Este módulo representa toda la configuración que tiene Jhipster para la generación de los archivos y la cual extiende y usa el generador de pipelines. Este módulo ayuda a inicializar algunas variables y constantes, precargar algunas configuraciones, cargar los prompts y generar los archivos basados en las respuestas.
- **Jhipster ci-cd:** Esta carpeta contiene los archivos que tienen la configuración que permite la ejecución del generador de pipelines.
  - **index.mts:** Este archivo es la puerta de entrada al flujo del generador de pipelines, el cual expone el archivo generator para continuar con el proceso.
  - **generator.mjs:** Este archivo contiene gran parte de la lógica del generador de pipelines, importando y usando los componentes del módulo base Jhipster. Adicionalmente,

importa los prompts a presentar al usuario, transmite las respuestas hacia las plantillas y define qué archivos se van a generar según la solicitud del usuario.

- **prompts.mjs:** Se mapean las diferentes preguntas y flujos que se van a presentar al usuario según las elecciones que ellos tomen en el proceso.

- **templates:** Esta carpeta que se encuentra dentro de Jhipster ci-cd, contiene todas las plantillas, con las cuelas se va a generar los diferentes pipelines. En esta, los archivos "azure-pipelines.yml.ejs", "github-actions.yml.ejs", "travis.yml.ejs", "circle.yml.ejs" y "circle.yml.ejs", son la plantilla respectiva para generar los pipelines en cada una de las herramientas, dentro de ellos se encuentran las diferentes validaciones para generar los pipelines con pruebas, entrega continua y/o depliegue continuo.

  El Archivo docker-registry.ejs, es un archivo de configuración de Docker Compose, que se utiliza para definir y ejecutar aplicaciones Docker multi-contenedor. Este archivo de Docker Compose permite ejecutar un registro de Docker y una interfaz de usuario asociada en contenedores Docker de manera sencilla y coordinada. Sin embargo, este archivo no fue modificado en el presente trabajo.

- **manifests:** Esta carpeta contiene dos archivos de manifiesto, los cuales se usan en los casos que se va a generar un pipeline que contenga la configuración para desplegar en Azure Kubernetes Service.

  - **deployment.yml.ejs:** Este archivo contiene la configuración necesaria de despliegue en un AKS, la cual contiene la información de la aplicación a desplegar y el contenedor donde se encuentra.
  - **service.yml.ejs:** En este archivo se configura el balanceador de carga, el cual va a ser la puerta de entrada hacia la aplicación desplegada en el AKS.

- **jenkinsdec:** Esta carpeta contiene los archivos a generar cuando el usuario decide crear un pipeline de Jenkins con el tipo declarativo.

  - **Jenkinsfile.ejs:** Este archivo es la plantilla que contiene la configuración para generar el Jenkinsfile, el cual es el pipeline que usa Jenkins. En este se realizan las validaciones necesarias para generar el pipeline adecuado según las elecciones del usuario.
  - **jenkins.yml.ejs:** Este archivo contiene la información necesaria para desplegar Jenkins mediante un contenedor docker.

- **jenkinsascp:** Esta carpeta contiene los archivos a generar cuando el usuario decide crear un pipeline de Jenkins con el tipo scripting en Groovy.

  - **Jenkinsfile.ejs:** Este archivo es la plantilla que contiene la configuración para generar el Jenkinsfile, el cual es el pipeline que usa Jenkins. En este se realizan las validaciones necesarias para generar el pipeline adecuado según las elecciones del usuario.

- **jenkins.yml.ejs:** Este archivo contiene la información necesaria para desplegar Jenkins mediante un contenedor docker.
- **idea.gdsl.ejs:** Este archivo es un script de definición de lenguaje Groovy (GDSL, por sus siglas en inglés) específicamente diseñado para proporcionar autocompletado y ayuda en IntelliJ IDEA cuando se trabaja con un Jenkinsfile en Groovy.

4.2.3.2. Implementación de variabilidad

Adicionalmente, como se describió en la subsección 4.1.6.1, para la implementación de la variabilidad del generador de pipelines se utilizan los siguientes enfoques:

- **Tiempo de Enlace (Binding Time):**
  - **Tiempo de compilación (variabilidad estática):** El generador utiliza principalmente el tiempo de compilación para implementar la variabilidad. Esto significa que el desarrollador selecciona las características que desea incluir durante la generación del pipeline, y el generador crea el código correspondiente solo para esas características. Las características no seleccionadas no se compilan ni se incluyen en el proyecto final.
- **Tecnología:** En este caso, el generador usa los dos enfoques de tecnología de la siguiente forma:
  - **Basado en el lenguaje de programación:** En parte, utiliza el lenguaje de programación Javascript para realizar las diferentes validaciones, respecto a las opciones que se le presentarían al usuario cuando se esté escogiendo las características. Además, de las características que irán en cada producto.
  - **Basado en la herramienta:** Adicional a las configuraciones que se realizan con el uso del lenguaje de programación, el generador usa Yeoman[19] para controlar el proceso de generación de los diferentes archivos.
- **Representación:**
  - **Anotación:** El generador utiliza un sistema de anotaciones para definir las características variables en el proyecto. Estas anotaciones permiten al desarrollador especificar qué características desea incluir y cómo deben implementarse. Así, el generador utiliza estas anotaciones para generar el código correspondiente a las características seleccionadas.

Cabe mencionar, que estos mismos enfoques son los que utiliza el generador de Jhispter para sus generadores, por tal razón se opta por ellas, para que así se acople de la mejor manera a su funcionamiento.

[19] https://yeoman.io/

4.2.3.3. Flujo de preguntas en el prompt

La interacción del usuario con el generador de pipelines, se realiza a través de una consola de comandos usando un prompt. Para iniciar el flujo se ejecuta el comando "*jhipster ci-cd*". Luego de este, se empiezan a realizar las diferentes preguntas, pero dependiendo del flujo y las selecciones que se hagan, aparecerán más o menos preguntas y más o menos opciones.

A continuación, se presentan estás preguntas:

- **What CI/CD pipeline do you want to generate?:** Esta pregunta tiene las siguientes opciones:
  - Jenkins declarative pipeline
  - Jenkins scripted pipeline
  - Azure Pipelines
  - Gitlab CI
  - GitHub Actions
  - Travis CI
  - Circle CI

Si se selecciona "*Jenkins scripted pipeline*" se presentan las siguientes dos preguntas adicionales al resto de las herramientas, las cuales son de sí y no.

- **Would you like to perform the build in a Docker container ?**
- **Would you like to send build status to GitLab ?**

Luego de seleccionar la herramienta de CI/CD y en el caso de seleccionar "*Jenkins scripted pipeline*" haber respondido las dos preguntas anteriores, se seleccionan qué integraciones se quieren incluir en el pipeline, en este caso, no todas las integraciones están disponibles para todas las herramientas de CI/CD. A continuación, se presentan las opciones para esta pregunta según la herramienta seleccionada.

- **What tasks/integrations do you want to include?:** En este caso, las opciones varían dependiendo de la herramienta de CI/CD seleccionada en el paso anterior. A continuación, se presentan las opciones, de las cuales se puede seleccionar las que se desee incluir. En el caso que no se seleccione ninguna, :
  - Jenkins declarative pipeline
    - *Snyk*: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)
    - Analyze your code with *Sonar*
    - Add deliver and deploy to the pipeline
  - Jenkins scripted pipeline

    - *Snyk*: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)
    - Analyze your code with *Sonar*
    - Add deploy on Heroku services to the pipeline
- Azure Pipelines
    - *Snyk*: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)
    - Add deliver and deploy to the pipeline
- Gitlab
    - *Snyk*: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)
    - Analyze your code with *Sonar*
    - Add deliver and deploy to the pipeline
- GitHub Actions
    - *Snyk*: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)
    - Analyze your code with *Sonar*
    - Add deliver and deploy to the pipeline
    - Would you like to enable the *Cypress Dashboard* (requires both CYPRESS PROJECT ID and CYPRESS RECORD KEY set on CI service)
- Travis CI
    - *Snyk*: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)
    - Analyze your code with *Sonar*
    - Add deliver and deploy to the pipeline
- Circle CI
    - *Snyk*: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)
    - Add deliver and deploy to the pipeline

Si se selecciona la opción "*Analyze your code with *Sonar**" se le presentará la pregunta al usuario "*Sonar*: what is the name of the Sonar server?" y por defecto se le dará la opción de escoger la palabra *sonar*.

Si al seleccionar "*Jenkins scripted pipeline*" se selecciona la opción "*Add deploy on Heroku services to the pipeline*", se le preguntará al usuario "**Heroku*: name of your Heroku Application?*" y por defecto se da la opción de escoger el nombre del proyecto.

Al seleccionar la opción "*Add deliver and deploy to the pipeline*" con cualquiera de las herramientas, se presentará el siguiente flujo de preguntas:

- **Where do you want to deliver or deploy the app?**
    - Azure: Deliver and/or deploy with azure services: Al seleccionar esta opción se presenta la pregunta "*What steps do you want to add to the pipeline?*" y brinda las siguientes opciones:

- only deliver: Al seleccionar esta opción se presenta la pregunta "How do you want to deliver the application?" y se dan las siguientes opciones:
  - Docker Image in Azure Container Registry
  - Jar file in the pipeline tool
- deliver and deploy: Al seleccionar esta opción, se pregunta "*Where do you want to deploy the application?*" y se presentan las siguientes opciones:
  - Azure Web Service
  - Azure Kubernetes Service
  - Azure Web Service for Containers

- Heroku: Deploy your application on Heroku: Si se selecciona esta opción, se realiza la siguiente pregunta:
  - name of your Heroku Application ?: Para esta opción, se da como opción por defecto el nombre del proyecto.

Al final se pregunta al usuario cual branch desea usar con el pipeline a generar, para así configurarlo en el trigger. La pregunta es:

- What *branch* do you want to publish: Por defecto se da la opción de escoger la palabra *master*

En las figuras 4.7 y 4.8 se presenta fragmentos de código donde se observa algunas de las configuraciones del prompt descrito previamente.

```
const prompts = [
  {
    type: 'list',
    name: 'pipeline',
    message: 'What CI/CD pipeline do you want to generate?',
    default: 'jenkinsdec',
    choices: [
      { name: 'Jenkins declarative pipeline', value: 'jenkinsdec' },
      { name: 'Jenkins scripted pipeline', value: 'jenkinsscp' },
      { name: 'Azure Pipelines', value: 'azure' },
      { name: 'GitLab CI', value: 'gitlab' },
      { name: 'GitHub Actions', value: 'github' },
      { name: 'Travis CI', value: 'travis' },
      { name: 'CircleCI', value: 'circle' },
    ],
  },
];
const props = await this.prompt(prompts);
this.pipeline = props.pipeline;
```

Figura 4.7: Fragmento uno de código de promtp.

```
const integrationChoices = [];
if (['jenkinsscp', 'jenkinsdec', 'gitlab', 'travis', 'github', 'circle', 'azure'].includes(this.pipeline)) {
    integrationChoices.push({
      name: `${chalk.yellow('*Snyk*')}: dependency scanning for security vulnerabilities (requires SNYK_TOKEN)`,
      value: 'snyk',
    });
  }
if (['jenkinsscp', 'jenkinsdec', 'gitlab', 'travis', 'github'].includes(this.pipeline)) {
  integrationChoices.push({ name: `Analyze your code with ${chalk.yellow('*Sonar*')}`, value: 'sonar' });
}
if (['jenkinsdec', 'travis', 'github', 'circle'].includes(this.pipeline)) {
  integrationChoices.push({
    name: `Add deliver and deploy to the pipeline`,
    value: 'deliverDeploy'
  })
} else if (['jenkinsscp', 'gitlab'].includes(this.pipeline)) {
  integrationChoices.push({
    name: `Add deploy on Heroku services to the pipeline`,
    value: 'heroku'
  })
} else {
  integrationChoices.push({
    name: `Add deliver and deploy on Azure services to the pipeline`,
    value: 'deliverDeployAz'
  })
}
```

Figura 4.8: Fragmento dos de código de promtp.

La obtención de los pipelines deseados, luego de tener las características seleccionadas por el usuario, se da gracias a la implementación de variabilidad con ayuda del lenguaje de programación y las herramientas adicionales, como Yeoman. En las figuras 4.9 y 4.10 se observa algunos fragmentos de código de la implementación realizada para permitir la variabilidad de los pipelines.

```
if (this.pipeline === 'gitlab') {
  this.writeFile('.gitlab-ci.yml.ejs', '.gitlab-ci.yml');
}
if (this.pipeline === 'circle') {
  this.writeFile('circle.yml.ejs', '.circleci/config.yml');
}
if (this.pipeline === 'travis') {
  this.writeFile('travis.yml.ejs', '.travis.yml');
}
if (this.pipeline === 'azure') {
  this.writeFile('azure-pipelines.yml.ejs', 'azure-pipelines.yml');
}
if (this.pipeline === 'github') {
  this.writeFile('github-actions.yml.ejs', '.github/workflows/main.yml');
}
if (this.deployService === 'aks') {
  this.writeFiles({
    rootTemplatesPath: 'manifests',
    blocks: [
      {
        templates: [
          {
            sourceFile: 'deployment.yml',
            destinationFile: '.manifests/deployment.yml'
          },
          {
            sourceFile: 'service.yml',
            destinationFile: '.manifests/service.yml'
          }
        ]
      }
    ],
    context: this
  })
}
```

Figura 4.9: Fragmento de código de variabilidad de pipelines.

```
<%_ if (buildToolMaven) { _%>
- script: chmod +x mvnw
<%_ } else if (buildToolGradle) { _%>
- script: chmod +x gradlew
<%_ } _%>
- task: Docker@2
  displayName: Login to Azure Container Registry
  inputs:
    command: login
    containerRegistry: $(CONTAINER_REGISTRY)
<%_ if (buildToolMaven) { _%>
- script: './mvnw package -Pprod verify jib:build -Djib.to.image=$(CONTAINER_REGISTRY)/$(IMAGE_REPOSITORY):$(TAG)'
  displayName: 'Build and package the application'
<%_ } else if (buildToolGradle) { _%>
- script: './gradlew jib -Pprod -Djib.to.image=$(CONTAINER_REGISTRY)/$(IMAGE_REPOSITORY):$(TAG)'
  displayName: 'Build and package the application'
<%_ } _%>
<%_ } _%>
```

Figura 4.10: Fragmento de código de variabilidad en un pipeline.

4.2.4. Derivación de Producto

Teniendo en cuenta las 70 configuraciones posibles, según el modelo de características descrito en la sección 4.2.1 y el desarrollo realizado en la sección 4.2.3, esto nos da la cantidad de productos disponibles en el generador, siendo de la misma manera 70 productos. Basado en lo anterior, se puede mencionar que por cada herramienta de CI/CD, hay 14 productos que se pueden generar, los cuales varían según la herramienta de compilación, el tipo de empaquetado, si se agrega las fases de entrega y despliegue y qué tipo de entrega y/o despliegue se seleccione. A continuación, se describen las características que tienen los pipelines en sus diferentes etapas, agrupándolos por la herramienta de CI/CD. Cabe mencionar que esta sección se enfoca en presentar los productos disponibles por la línea de productos de software y no en las pruebas, ya que, una sección posterior se encarga de esta tarea.

4.2.4.1. Azure Pipelines

- **Configuración general:**

  - Trigger: Inicialmente se configura cuándo se va a ejecutar el pipeline, para ello, mediante el comando trigger se indica la rama en el repositorio del proyecto, la cual va a disparar la ejecución del pipeline cuándo haya un nuevo commit.
  - Variables: Adicionalmente, se configuran variables a utilizar en el proceso del pipeline, de las cuales, algunas pueden ser para configuración del ambiente y otras para acceder a los recursos en la nube, por lo que se deben mantener seguras. Para esas variables sensibles, dentro del pipeline se crea la referencia hacia variables que deben ser creadas como variables del pipeline, utilizando la interfaz de usuario, azure cli o las diferentes opciones que Azure tiene. En la sección de variables del pipeline y en el pipeline en general es posible encontrar los nombres con los cuales se debe definir estos campos,

como lo son la subscripción o el nombre del servicio en el que se va a desplegar, usando el formato $(<Nombre de la variable>).

- Stages: Se configuran 3 etapas, la primera Test, se enfoca en ejecutar las diferentes pruebas disponibles de la aplicación, como unitarias y de integración, esta sección se basa en los comandos ya incluidos en el package.json de la aplicación y el pipeline generado por el generador de jhipster. La segunda sección se enfoca en la entrega del artefacto o contenedor luego de haber superado todas las pruebas. Y la última etapa en desplegar la aplicación en el servicio correspondiente. Es importante tener en cuenta que en la primera y segunda etapa se configura la versión de Java a utilizar, para evitar errores de compatibilidad.

- **JAR a Web App:** Como se mencionó anteriormente, se crean las tres etapas de Test, Build y Deploy. Para la etapa Test, se utiliza el pipeline que brinda el generador de Jhipster pero configurando la versión de Java adecuada. Para la segunda etapa se configura nuevamente la versión de Java adecuada, luego se construye el archivo JAR de la aplicación y se carga el artefacto al directorio de artefactos del pipeline. Finalmente, para la tercera etapa de despliegue, se configura la información correspondiente a la Web App a utilizar y el artefacto a desplegar.

- **Docker a Web App for Containers:** Para la primera etapa, se usa la misma configuración del anterior Pipeline. Sin embargo, para la segunda etapa, aunque se mantiene la configuración de la versión de Java, se cambia el archivo a generar y la forma de entregar, ya que, en este caso se genera un contenedor docker con la aplicación y se carga en Azure Container Registry, para el cual se deben configurar las llaves necesarias. Y para la tercera etapa, se configura la información correspondiente al Web App for Container a utilizar y el archivo docker a desplegar. Un detalle para tener en cuenta en este proceso, son los permisos con los que debe contar el pipeline en Azure Pipelines y el Azure Web App for Container para acceder a Azure Container Registry y el contenedor que se cargue a él.

- **Docker a Azure Kubernetes Service:** La primera etapa se mantiene sin cambios, respecto a los anteriores pipelines. Para la segunda etapa se configura nuevamente la versión de Java adecuada y se construye y carga el contenedor docker de la aplicación a Azure Container Registry. Es importante mencionar, que en este caso se deben generar dos archivos más, aparte del archivo yml del pipeline, los cuales son la configuración de los contenedores y del balanceador de carga. Estos archivos se deben subir también. Para la tercera etapa, se debe configurar la información correspondiente a Azure Kubernetes Service y los secretos y conexiones necesarias a utilizar, además de la imagen docker a desplegar.

En la figura 4.11 se observa un fragmento del stage de deliver para desplegar en Azure Kubernetes Service con Azure pipelines.

```yaml
- stage: Deliver
  displayName: Deliver stage
  jobs:
    - job: deliver
      displayName: publish artifact or docker
      pool:
        vmImage: $(VM_IMAGE_NAME)

      steps:
        - task: JavaToolInstaller@0
          inputs:
            versionSpec: 17
            jdkArchitectureOption: 'x64'
            jdkSourceOption: 'PreInstalled'
          displayName: 'Change Java version'
        - task: Npm@1
          inputs:
            command: 'custom'
            customCommand: ' run java:jar:prod'
          displayName: 'TESTS: packaging'

        - task: CopyFiles@2
          displayName: 'Copy Files to artifact staging directory'
          inputs:
            SourceFolder: '$(System.DefaultWorkingDirectory)'
            Contents: '**/build/libs/*.jar'
            TargetFolder: $(Build.ArtifactStagingDirectory)

        - upload: $(Build.ArtifactStagingDirectory)
          artifact: drop
```

Figura 4.11: Fragmento de pipeline en Azure Pipelines.

4.2.4.2. Github Actions

- **Configuración general:**

  - permissions: Se agregan permisos para que desde el pipeline se pueda leer y modificar los archivos necesarios en el proceso de ejecución del pipeline.
  - name: El nombre del pipeline que va a aparecer en Github Actions
  - on: En este atributo se configura cuando se va a ejecutar el pipeline, por ejemplo, al realizar un push o un pull request sobre una rama definida.
  - env: En este atributo se definen las diferentes variables a utilizar en el pipeline.
  - jobs: Se configuran 3 jobs, el primero para las pruebas llamado Test, se enfoca en ejecutar las diferentes pruebas disponibles en la aplicación, tanto unitarias como de integración,

utilizando algunos de los comandos ya incluidos en el package.json de la aplicación y el pipeline generado por el generador de Jhipster. La segunda sección se enfoca en la entrega del artefacto o contenedor luego de haber superado todas las pruebas. Y la última etapa en desplegar la aplicación en el servicio correspondiente. Es importante tener en cuenta que en la primera y segunda etapa se configura la versión de Java a utilizar, para evitar errores de compatibilidad.

- **JAR a Web App:** Como se mencionó anteriormente, se crean las tres etapas de pruebas, entrega y despliegue de la aplicación. Para el primer job se utiliza el pipeline generado por Jhipster y se adecua a la versión de Java correspondiente. En el segundo job de entrega de la aplicación, se carga el código del repositorio al pipeline, se configura el Java 17 y posteriormente se genera y carga el artefacto al almacenamiento del pipeline. Por último, en el tercer job de despliegue, se configura el ambiente en el que se va a desplegar la aplicación, luedo se descarga el artefacto que se cargó en el anterior job, posteriormente se inicia sesión en Azure usando las credenciales necesarias y finalmente se despliega el artefacto en el Web App Service.

- **Docker a Web App for Containers:** Para el primer job se usa la misma configuración del anterior Pipeline. Luego en el segundo job, el objetivo es cargar el contenedor docker de la aplicación a Azure Repository, para ello, se carga el código del proyecto al pipeline para poder acceder a sus archivos, se configura la versión de Java, se inicia sesión en Azure y finalmente se genera y carga el archivo al repositorio.

- **Docker a Azure Kubernetes Service:** Para este pipeline, el primer y segundo job se mantienen sin cambios, respecto a los anteriores pipelines. Y el último job, se encarga de tomar la imagen de la aplicación desde Azure Container Registry y desplegarlo en el Azure Kubernetes Service, para ello, se inicia sesión en azure, se configura el contexto del AKS y el kubectl, para finalmente, desplegar la imagen, con los archivos del manifest que configuran los servicios y el balanceador de carga.

En la figura 4.12 se observa un fragmento del stage de deliver del pipeline para desplegar en Azure Web App con Github Actions.

```yaml
deliver:
  needs: test
  runs-on: ubuntu-latest
  steps:
    - name: Checkout code
      uses: actions/checkout@v3
    - name: Maven Package
      uses: actions/setup-java@v3
      with:
        distribution: 'temurin'
        java-version: 17
    - name: Package application
      run: npm run java:jar:prod
    - name: Publish Artifact
      uses: actions/upload-artifact@v3
      with:
        name: drop
        path: ${{ github.workspace }}/build/libs/*.jar
```

Figura 4.12: Fragmento de pipeline en Github Actions.

4.2.4.3. CircleCI

- **Configuración general:**
  - workflows: En el caso de circle ci, en este atributo se definen los jobs que se van a ejecutar, configurando si se requiere la ejecución primero de uno de ellos para la ejecución del otro o si se pueden ejecutar al tiempo sin problema, lo ideal es que la entrega requiera la ejecución completa de las pruebas y el despliegue la ejecución de la entrega. Adicionalmente, aquí se puede definir que rama del repositorio será la que se va a ejecutar al haber un nuevo commit.

- **JAR a Web App:** Para este pipeline se definen tres jobs, en el primero se realizan las pruebas unitarias y de integración, para lo que se necesita acceder a los archivos del proyecto dentro del pipeline y configurar la versión de Java. Posteriormente, se crea el job para la entrega del artefacto JAR, para lo cual, se configura nuevamente la versión de Java, se crea el archivo JAR y posteriormente se almacena este en el workspace para poder usarlo en los otros jobs. Al final, en el último job se realiza el despliegue de la aplicación, para ello, se configuran variables para así definir llaves a usar, luego se usa como entorno una imagen que

contiene azure-cli configurado, para así, en este adjuntar el workspace y configurar el inicio de sesión y despliegue en el Web App Service del artefacto.

- **Docker a Web App for Containers:** Nuevamente se divide el pipeline en tres jobs, de los cuales, el primero es igual al anterior pipeline. Y para el segundo job, en este caso, se configuran las variables a utilizar, se agrega el proyecto al workspace, luego se configura la versión de Java, para así iniciar sesión en azure y generar la imagen docker y subirla a Azure Container Registry.
- **Docker a Azure Kubernetes Service:** Se mantienen nuevamente los primeros jobs. Para el último job de despliegue en cambio, se definen todas las variables necesarias y luego se instalan az-cli y kubectl, ya que, en este caso se usa imagen de ubuntu como entorno para el job. Al tener el ambiente listo, se inicia sesión en Azure, se configura el contexto del AKS y se despliega la aplicación.

En la figura 4.13 se observa un fragmento del stage de deliver del pipeline para hacer solo entrega en Azure Container Registry con Circle CI.

```
deliver:
  machine:
    image: ubuntu-2004:current
  steps:
    - checkout
    - run:
        name: Install Java 17
        command: |
          sudo apt-get update && sudo apt-get install -y openjdk-17-jdk
    - run:
        name: Give Executable Power
        command: 'chmod +x gradlew'
    - run:
        name: Login to Azure Container Registry
        command: |
          echo $CONTAINER_REGISTRY_PASSWORD | docker login $CONTAINER_REGISTRY --username $CONTAINER_REGISTRY_USERNAME --password-stdin
    - run:
        name: Build and package the application
        command: |
          ./gradlew -Pprod bootJar jib -Djib.to.image=$CONTAINER_REGISTRY/$IMAGE_REPOSITORY:$CIRCLE_WORKFLOW_ID
```

Figura 4.13: Fragmento de pipeline en Circle CI.

4.2.4.4. Travis-CI

- **Configuración general:**
  - branches: Se configura las ramas a las que se va a restringir y desde las cuaes se va a ejecutar el pipeline, cada vez que haya un commit.
  - services: Se declaran los servicios que se utilizarán, en este caso, Docker.
  - language: Se establece el lenguaje que se utilizará para la construcción y las pruebas (16.17.0).
  - node_js: La versión de node que se utilizará en el proceso (16.17.0).

- cache: Se configura el caché para directorios como node, node_modules y $HOME/.m2 para mejorar la velocidad de compilación en futuras ejecuciones.

- env: Se definen varias variables de entorno globales que se utilizarán en todo el pipeline, como las credenciales de Azure y la información relacionada con Docker y Kubernetes

- stages: En este caso, el pipeline varía entre dos y tres etapas: test (pruebas), deliver (entrega) y deploy (despliegue).

- **JAR a Web App:** En este caso se crean dos etapas, uno para las pruebas y el otro para el despliegue. La primera etapa utiliza el pipeline generado por Jhipster, con las pruebas de back y front proporcionadas, pero configurando la versión de Java necesaria. En la segunda etapa se realiza la entrega y el despliegue de la aplicación, ya que, a diferencia de los anteriores pipelines, en Travis CI no se puede guardar artefactos en el workspace del pipeline, por lo que se debe crear el archivo JAR y desplegar la aplicación en el mismo stage, para ello, se configura de igual forma la versión de Java, se instala azure cli y luego se procede a iniciar sesión en Azure para así desplegar la aplicación en Web App Service.

- **Docker a Web App for Containers:** En este caso, se crean las tres etapas. La primera se mantiene igual que la anterior. La segunda etapa en este caso se enfoca en la carga del contenedor y la tercera etapa en el despliegue de ese contenedor en el App Service.

- **Docker a Azure Kubernetes Service:** La primera etapa se mantiene sin cambios, respecto a la anterior descripción y en el tercera etapa se configura al Azure Kubernetes Service para desplegar el contenedor en este servicio.

En la figura 4.14 se observa un fragmento del stage de deploy, para desplegar la aplicación en Azure Web App con Travis CI.

```
- stage: deploy
  name: Build, package and deploy
  services:
    - docker
  before_script:
    - |
      if [[ $JHI_JDK = '17' ]]; then
        echo '*** Using OpenJDK 17'
        sudo add-apt-repository ppa:openjdk-r/ppa -y
        sudo apt-get update -q
        sudo apt-get install -y openjdk-17-jdk -y
        sudo update-java-alternatives -s java-1.17.0-openjdk-amd64
        export JAVA_HOME=/usr/lib/jvm/java-17-openjdk-amd64
        export PATH=$PATH:$JAVA_HOME/bin
      else
        echo '*** Using OpenJDK 11 by default'
        sudo apt-get update -q
        sudo apt-get install -y openjdk-11-jdk
        sudo update-java-alternatives -s java-1.11.0-openjdk-amd64
        export JAVA_HOME=/usr/lib/jvm/java-11-openjdk-amd64
        export PATH=$PATH:$JAVA_HOME/bin
      fi
      java -version
    - sudo /etc/init.d/mysql stop
    - sudo /etc/init.d/postgresql stop
    - curl -sL https://aka.ms/InstallAzureCLIDeb | sudo bash
    - nvm install $NODE_VERSION
    - npm install -g npm
    - npm install
    - az login --service-principal -u ${AZURE_CLIENT_ID} -p ${AZURE_CLIENT_SECRET} --tenant ${AZURE_TENANT_ID}
  script:
    - npm run java:jar:prod
    - az webapp deploy --resource-group ${RESOURCE_GROUP} --name ${WEBAPP_NAME} --src-path $(find . -name "*.jar") --type jar --verbose
```

Figura 4.14: Fragmento de pipeline en Travis CI.

4.2.4.5. Jenkins

- **Configuración general:**

  - Sintaxis: Teniendo en cuenta que en Jenkins existen dos formas de crear pipelines, 'declarative' y 'scripted', y aunque las dos pueden realizar las mismas tareas, se decide utilizar la forma declarativa, ya que, como lo describe la documentación de Jenkins, este posee características sintácticas más ricas y está diseñado para ser más fácil de leer y escribir [20].
  - Ambiente: Debido a que en Jhipster se realizan las pruebas utilizando contenedores docker, lo mejor es no utilizar Jenkins en un contenedor, ya que el ejecutar contenedores dentro de contenedores, siempre acarrea problemas complicados de controlar, por lo que se instala Jenkins localmente en el sistema a ejecutar el pipeline. Debido a esto, la versión de Java, azure-cli o kubectl, que el pipeline utilice serán las herramientas que se encuentren instalas en el computador.
  - agent: en este atributo se define que agente se desea utilizar al ejecutar el pipeline, al utilizar el término any, se toma el primer agente disponible y en este caso, sería el computador en que se encuentra instalado Jenkins.

[20] https://www.jenkins.io/doc/book/pipeline/

- environment: Se definen las diferentes variables a utilizar en la ejecución del pipeline, como las llaves de Azure.

- **JAR a Web App:** Aunque Jhipster genera una primera versión, que realiza las pruebas y entrega un archivo JAR, este pipeline utiliza la sintaxis 'scripted', por lo que se cambia a la sintaxis declarativa, creando las etapas de pruebas y entrega y se crea una nueva etapa de despligue donde se inicia sesión en Azure y se despliega la aplicación en Azure Web App.
- **Docker a Web App for Containers:** La primera etapa del pipeline anterior se mantiene para las pruebas. Para la segunda etapa de entrega, se crea y se carga la imagen de la aplicación en Azure Container Registry. Y en la última etapa se despliega esta imagen en el Web App Service para contenedores.
- **Docker a Azure Kubernetes Service:** Para este pipeline, se mantienen las primeras dos etapas y en la última etapa de despliegue se inicia sesión en Azure, se configuran las credenciales para el Azure Kubernetes Service y se cargan y despliegan la imagen y los servicios junto al balanceador de carga en el AKS.

En la figura 4.15 se observa un fragmento del stage de deploy, para desplegar en Azure Kubernetes Service con Jenkins.

```
stage('Deliver') {
    steps {
        script {
            checkout scm
            sh "chmod +x gradlew"
            sh "./gradlew jib -Pprod -Djib.to.image=\$CONTAINER_REGISTRY/\$IMAGE_REPOSITORY:\$BUILD_NUMBER"
        }
    }
}

stage('Deploy') {
    agent any
    steps {
        checkout scm
        sh "az login --service-principal --username \$AZURE_CLIENT_ID --password \$AZURE_CLIENT_SECRET --tenant \$AZURE_TENANT_ID"
        sh "echo \"\${CONTAINER_REGISTRY_PASSWORD}\" | docker login --username \${CONTAINER_REGISTRY_USERNAME} --password-stdin \${CONTAINER_REGISTRY}"
        sh "az login --service-principal --username \$AZURE_CLIENT_ID --password \$AZURE_CLIENT_SECRET --tenant \$AZURE_TENANT_ID"
        sh "az aks get-credentials --resource-group \"\$AZURE_RESOURCE_GROUP\" --name \"\$CLUSTER_NAME\""
        sh '''
            kubectl apply -f deployment.yml -f service.yml
            kubectl set image deployment/$DEPLOYMENT_APP_NAME $CONTAINER_REGISTRY_USERNAME=$CONTAINER_REGISTRY/$IMAGE_REPOSITORY:$BUILD_NUMBER
        '''
        sh 'az logout'
    }
}
```

Figura 4.15: Fragmento de pipeline en Jenkins.

4.3. Repositorio del Generador de Pipelines de CI/CD

En el siguiente repositorio, "https://github.com/Robcruzd/generator-jhipster", se puede encontrar el código de la bifurcación del generador de Jhipster en su versión 7.9.3, el cual se sugiere usar para generar los proyectos. Y el generador de CI/CD al cual se le realizaron las mejoras descritas en el presente proyecto, se encuentra en el siguiente enlace del repositorio "https:

//github.com/Robcruzd/generator-jhipster/tree/main/generators/ci-cd", aquí se tiene un README con la descripción del proyecto y los comandos para hacer uso de él.

4.4. Resumen del capítulo

En la primera sección de este capítulo se definieron las diferentes herramientas y técnicas a utilizar en la implementación del generador de pipelines, como lo son las herramientas de CI/CD, los servicios de nube a utilizar para desplegar la aplicación, las líneas de productos de software como técnica de reutilización y variabilidad, entre otras definiciones.

En la segunda sección se presentó la implementación de la línea de productos de software para la obtención del generador de pipelines, llevando a cabo las cuatro fases descritas por esta técnica de reutilización y variabilidad, como lo son el análisis de dominio, análisis de requerimientos, implementación de dominio y derivación de producto.

Capítulo 5

Evaluación

En este capítulo, se presenta la evaluación del generador de pipelines para CI/CD desarrollado en el marco de este proyecto. Con el objetivo de medir la efectividad, la usabilidad y la adaptabilidad de la herramienta, se llevó a cabo un proceso de evaluación desde dos perspectivas:

Por un lado, se realizó una evaluación técnica mediante la implementación de pipelines en proyectos de prueba, seguido por el despliegue de una aplicación. Esta fase proporcionó información valiosa sobre la capacidad del generador para integrarse con diversas tecnologías y ejecutar procesos de CI/CD de manera correcta.

Por otro lado, se realizó una evaluación concreta desde la perspectiva de los desarrolladores de software. A través de un enfoque participativo, se invitó a desarrolladores a utilizar la herramienta, generar sus propios pipelines y desplegar aplicaciones de prueba. La retroalimentación recopilada durante esta fase se revela como un componente esencial para comprender la experiencia práctica de los usuarios con el generador.

Este capítulo describe los métodos utilizados para la evaluación, presenta los resultados obtenidos, y proporciona una visión integral de las recomendaciones y mejoras sugeridas por los usuarios.

5.1. Diseño de la evaluación

Considerando las pautas establecidas en “Evaluation Methodology Basics” (Davidson, 2005) y “Evaluation Methodology” (Baehr, 2004), se procedió a la definición e identificación clara de los objetivos de la evaluación. Se establecieron criterios específicos para la evaluación, se determinaron las fuentes de evidencia a utilizar y se llevó a cabo un proceso para la selección de la muestra. Para evitar sesgos subjetivos, se implementaron métricas y técnicas de ponderación, y se desarrollaron métodos para condensar y sintetizar de manera efectiva los hallazgos obtenidos durante el proceso de evaluación.

Para cubrir el objetivo 4 y 5 del presente proyecto, se realizaron dos tipos de evaluación. La primera se enfoca en evaluar el funcionamiento adecuado del generador de pipelines con tres casos de estudio que permitan simular la realidad. La segunda fase se centra en evaluar el uso de la herramienta por parte de desarrolladores de software, validando la satisfacción respecto al uso de la herramienta, que tan útil es para ellos, posibles mejoras sobre la herramienta y los pipelines en sí, entre otros, que se definen en la sección de preguntas de investigación.

5.1.1. Definición de la evaluación - Fase 1

La primera fase se enfoca en el objetivo número 4, en el cual se busca evaluar la herramienta utilizando algunos proyectos de prueba. Para lograr este objetivo, se generaron 2 proyectos utilizando Jhipster, con características diferentes y utilizando un esquema de base de datos que simule la realidad. Adicionalmente, se descargó y utilizó un proyecto disponible en la página de Jhipster.

5.1.1.1. Propósito de la evaluación

El propósito de la evaluación es identificar el funcionamiento y compatibilidad del generador usando algunos proyectos de prueba y llevándolos hasta el despliegue en Azure.

5.1.1.2. Preguntas de investigación a responder durante la evaluación

En esta sección, se plantean preguntas de investigación clave que guiarán la evaluación con proyectos de prueba del generador de pipelines. Estas preguntas están diseñadas para profundizar en aspectos específicos del funcionamiento y compatibilidad del generador y su interacción con proyectos de prueba. La obtención de respuestas detalladas a estas interrogantes contribuirá a una comprensión más precisa de los resultados obtenidos en esta fase de la evaluación.

- ¿El generador de pipelines se adapta a las tecnologías presentes en los proyectos de prueba?
- ¿Es posible generar todas las configuraciones posibles para cada uno de los proyectos sin errores?
- ¿Los pipelines generados con la herramienta se ejecutan exitosamente en las herramientas de CI-CD?
- ¿El generador crea los archivos necesarios para el despliegue en los diferentes servicios?
- ¿El generador crea únicamente el código relacionado a las configuraciones realizadas a través del prompt?
- ¿Se presentan únicamente las preguntas y respuestas posibles dependiendo de las respuestas previas en el flujo del prompt?

5.1.1.3. Selección de la muestra

Para seleccionar los proyectos con los cuales se va a realizar la validación del funcionamiento del generador de pipelines, se utilizaron los siguientes criterios:

- El proyecto debe haber sido generado usando Jhipster
- El proyecto generado debe ser un monolito con Spring boot
- El proyecto debe ser funcional
- La versión de Jhipster utilizada para generar el proyecto debe ser la 7

5.1.1.4. Unidades de Análisis

Para realizar las pruebas se seleccionaron tres proyectos para poder validar las preguntas de investigación de la subsección 5.1.1.2. A continuación se describen cada uno de ellos:

- **Proyecto Jhipster con Maven:** Utilizando Jhipster, se generó un proyecto con las características descritas en la subsección 4.1.1, pero escogiendo como compilador a Maven. Adicionalmente, se agregó un diagrama de entidades por defecto que tiene la herramienta de Jhipster JDL Studio[1], como se observa en la figura 5.1, relacionado con un historial de trabajos. Al tener el archivo jdl, con el generador de Jhipster se crearon las entidades presentes en el modelo dentro del proyecto, esto con el objetivo de que el proyecto se asemeje a proyectos de desarrollo comunes que utilicen bases de datos y entidades con las cuales se pueda realizar un CRUD.

- **Proyecto Jhipster con Gradle:** Utilizando Jhipster, se generó otro proyecto con las características descritas en la subsección 4.1.1, pero escogiendo como compilador a Gradle. Además, al igual que el anterior proyecto se tomó el mismo modelo entidad relación de la figura 5.1, para que se asemeje a un proyecto real.

- **InfinityShopping:** Este es uno de los proyectos que se encuentra publicado en la página de Jhipster, en la sección de casos para mostrar, los cuales han sido enviados voluntariamente por los creadores hacia los administradores de Jhipster. Este proyecto se encuentra público en el siguiente enlace de github *https://github.com/PiotrZielonka/infinityshopping/tree/develop*. La idea de este proyecto es un marketplace para vender productos y está realizado con Maven.

5.1.1.5. Métricas definidas

Estás métricas son las unidades que se definen para realizar la medición durante la evaluación y así responder las preguntas de investigación de la subsección 5.1.1.2

- Errores de compatibilidad: Número de errores de compatibilidad por el stack de cada proyecto.
- Errores de generación: Número de errores en el proceso de generación del pipeline
- Errores en la ejecución: Número de errores al ejecutar el pipeline.
- Errores de creación: Número de errores respecto a los archivos creados.
- Errores en el código: Número de errores en el código, basado en las veces que sobró o faltó código según las configuraciones realizadas
- Errores en el prompt: Número de errores en el prompt, relacionados a las preguntas y los flujos de las preguntas.

[1] https://start.jhipster.tech/jdl-studio/

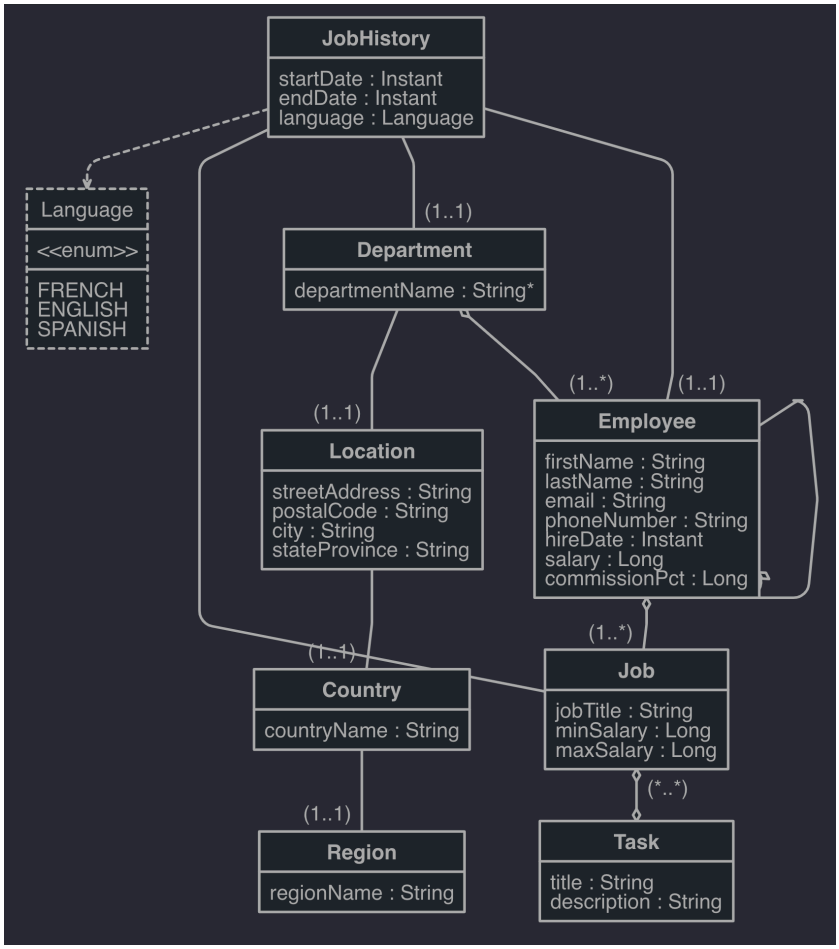


Figura 5.1: Diagrama Entidad Relación. Fuente JDL Studio Jhipster

5.1.1.6. Método de recolección de datos

Para realizar la evaluación se tomó cada uno de los proyectos y se generaron los pipelines posibles. En ese caso, se generaron 15 pipelines para cada proyecto y se realizaron despliegues en las diferentes Herramientas de CI/CD.

5.1.1.7. Resultados esperados

Al realizar esta evaluación se esperan los siguientes resultados:

- Posibles errores y correcciones por realizar en el funcionamiento del generador de pipelines.
- Encontrar posibles mejoras en el prompt y en el código del generador.
- Validar que los pipelines se generan y se ejecutan correctamente en las herramientas de CI/CD.

5.1.1.8. Riesgos de la validez de la evaluación

Basados en la revisión de Zhou et al. (2016), se tiene en cuenta que las evaluaciones tienen un riesgo de validez, para ello ha identificado cuatro categorías, las cuales, se deben tener en cuenta y se debería buscar la forma de mitigar o contrarrestar. A continuación, se presentan algunas de ellas relacionadas a la fase, con la respectiva estrategia en que se valida y por tanto se mitiga el riesgo:

- **Validez de constructo:** La idea de esta categoría es que la evaluación esté encaminada hacia lo que se desea estudiar.

  Riesgo: La evaluación podría no estar bien encaminada.

  Estrategia de mitigación: En este caso la estrategia inicial fue guiarse por el objetivo a cumplir en el presente trabajo de grado. Luego se revisó la definición de *funcionamiento*[2], para así tener claridad de lo que se buscaba encontrar, según este objetivo. Teniendo este concepto se analizó el generador de pipelines y sus diferentes puntos de falla.

- **Validez interna:** En este caso se valida qué tan precisa es la evaluación validando una relación causal, con la cual se cree que ciertas consideraciones conducen a otras.

  Riesgo: El generador y los pipelines podrían no funcionar correctamente.

  Estrategia de mitigación: Inicialmente asegurarse que se generan los pipelines correctamente y sin errores para las diferentes configuraciones y proyectos de prueba. Adicionalmente, se realizaron despliegues con los pipelines para validar su correcta ejecución y así mitigar los posibles problemas. En los dos casos se identificaron y corrigieron cualquier error o anomalía encontrada.

5.1.2. Definición de la evaluación - Fase 2

La fase 2 de la evaluación se enfoca en realizar entrevistas a desarrolladores de software que tengan experiencia trabajando con pipelines de CI/CD, y de esa manera validar la satisfacción y la utilidad del generador de pipelines para personas que estén en un ambiente laboral.

5.1.2.1. Propósito de la evaluación

El propósito de la evaluación es identificar la percepción de utilidad y satisfacción de los entrevistados al generar pipelines con la herramienta y observar el despliegue de la aplicación. Todo mediante la visión de personas que tengan conocimientos del uso de pipelines por su entorno laboral.

5.1.2.2. Preguntas de investigación a responder durante la evaluación

Con el objetivo de obtener respuestas más detalladas y exhaustivas al culminar la evaluación, se formularon preguntas de investigación abiertas que facilitan un análisis más profundo. Es crucial destacar que las fuentes utilizadas para abordar estas preguntas incluyen la evaluación inicial en los

[2] https://dle.rae.es/funcionamiento

proyectos de prueba, la recopilación de datos a través de encuestas dirigidas a los desarrolladores y la información derivada de la observación durante las entrevistas. Este enfoque integral, basado en distintas fuentes de datos, busca proporcionar una comprensión completa y matizada de los resultados obtenidos, contribuyendo así a una evaluación más rica y fundamentada.

A continuación, se presentan las preguntas que se busca responder con la evaluación:

- ¿Qué tan fácil es usar la herramienta en comparación con otras herramientas para crear pipelines?
- ¿Qué grado de compatibilidad hay entre el generador y diversas tecnologías del ambiente laboral?
- ¿El generador facilita el desplegar proyectos Java que se compilen con Gradle y o Maven?
- ¿Qué tan rápido es el generador, en comparación a procesos tradicionales?
- ¿Cuáles son las áreas de mejora y temas a corregir para la herramienta de generación de pipelines?
- ¿Qué otras herramientas similares hay y en que se diferencian?
- ¿En qué situaciones los desarrolladores usarían la herramienta?

5.1.2.3. Selección de la muestra

Para seleccionar las personas con las cuales se va a realizar la validación de utilidad del generador de pipelines, se utilizaron los siguientes criterios:

- Debe tener experiencia creando o modificando pipelines en su labor diaria con cualquier herramienta de CI-CD.
- Debe haber trabajado con proyectos compilados con Maven o Gradle.

5.1.2.4. Unidades de Análisis

Las personas seleccionadas para evaluar la herramienta son desarrolladores de software colombianos, que accedieron a participar en el proceso de evaluación y cumplieron los criterios definidos en la subsección 5.1.2.3. A continuación se describe a cada uno de ellos, por temas de privacidad no se menciona su nombre:

- **Participante 1:** Ingeniero de sistemas, con aproximadamente 7 años de experiencia, actualmente se desempeña como desarrollador backend en Bancolombia con proyectos Java utilizando Gradle y Maven. Y Durante su vida laboral ha tenido que trabajar modificando pipelines con Azure pipelines.

- **Participante 2:** Ingeniero de sistemas, con aproximadamente 7 años de experiencia que actualmente se desempeña como líder técnico en Nequi y ha trabajado mayormente en el backend en proyectos de diferentes tecnologías, incluyendo Java y Spring boot. Para crear y modificar pipelines ha utilizado herramientas como Jenkins y Github Actions.
- **Participante 3:** Ingeniero en eléctrica y electrónica, con más de 7 años de experiencia, que se desempeña como líder técnico en Nequi, ha liderado y desarrollado proyectos en diferentes tecnologías incluyendo Java y en su labor ha tenido que modificar pipelines con Jenkins.
- **Participante 4:** Ingeniero en electrónica y telecomunicaciones, con más de 14 años de experiencia, que se desempeña como líder técnico en Mercado libre, en su vida laboral ha liderado y desarrollado proyectos Java y ha tenido que crear y modificar pipelines usando Jenkins.
- **Participante 5:** Ingeniero en Automática industrial, con más de 10 años de experiencia, que trabaja como desarrollador de software en Globant, ha trabajado en varios proyectos en su vida laboral, liderando, desarrollando y coordinando. Con relación a herramientas de CI/CD ha trabajado con Github Actions.
- **Participante 6:** Ingeniero de sistemas con aproximadamente 20 años de experiencia, que actualmente se desempeña como gestor de producto técnico en Onurix, ha trabajado como desarrollador y líder técnico en proyectos backend. Como herramienta de CI/CD ha utilizado codepipeline para crear y editar pipelines.
- **Participante 7:** Ingeniero de sistemas e ingeniero en telemática con más de 7 años de experiencia que trabaja como ingeniero de software senior en Epam anywhere, desarrollando diferentes proyectos backend. Para crear y gestionar pipelines ha usado Jenkins y Github Actions.
- **Participante 8:** Ingeniero en electrónica y telecomunicaciones con más de 7 años de experiencia que trabaja en Endava como desarrollador de software senior. Ha desarrollado proyectos en el backend usando .Net y Java y como herramientas de CI/CD ha usado Azure pipelines.

5.1.2.5. Métricas definidas

Estás métricas son las unidades que se definen para realizar la medición durante la evaluación y así responder las preguntas de investigación de la subsección 5.1.2.2

- Percepción de utilidad: Este valor es dado por los desarrolladores según su percepción entre 1 y 10.
- Percepción de compatibilidad: Valor de compatibilidad según los desarrolladores entre 1 y 10.
- Percepción de rapidez: Valor de la percepción de rapidez de la herramienta según el desarrollador entre 1 y 10.

- Número de mejoras: Mejoras posibles a realizar a la herramienta.
- Número de correcciones: Correcciones necesarias a realizar a la herramienta.
- Número de herramientas similares: Cantidad de herramientas similares según la percepción de los desarrolladores.

5.1.2.6. Método de recolección de datos

Para llevar a cabo la recolección de datos, se hizo mediante reuniones con las personas mencionadas en la subsección 5.1.2.4 que cumplan los criterios de la subsección 5.1.2.3. Para ello se buscó a cada uno de ellos y se validó los criterios, y así luego acordar una cita. Para el desarrollo de la reunión el protocolo era el siguiente:

1. La reunión se llevaba a cabo de forma remota a través de una llamada de Zoom, por lo cual, el organizador enviaba un enlace para hacer la conexión.
2. Se daba contexto de la herramienta, las tecnologías y las técnicas utilizadas en el desarrollo de esta, al igual que el estado actual del ambiente.
3. Se hacía una demostración del uso de la herramienta generando un pipeline y se desplegaba la aplicación en Azure utilizando el pipeline creado.
4. Luego el ejercicio se realizaba ejecutando control remoto por parte del participante, ya que el organizador tenía el ambiente configurado para poder llevar a cabo la prueba. Los siguientes pasos se realizaban en el equipo remoto para el participante.
5. En Visual Studio Code, se tenían dos proyectos creados utilizando Jhipster, uno con Gradle y el otro con Maven, por lo que se debía escoger con cual se deseba proceder.
6. En el proyecto escogido, en la terminal y ubicado dentro del proyecto se debía ejecutar el comando “jhipster ci-cd”. Mediante este comando se iban habilitando las siguientes opciones de configuración dependiendo lo que se vaya escogiendo en cada paso, ya que algunas opciones son incluyentes o excluyentes de otras. Es importante tener en cuenta, que la idea era probar las configuraciones de entrega y despliegue continuo.
7. Al final de todos los pasos, se generaba un pipeline ubicado dentro del mismo proyecto, aunque su ubicación exacta dependía de la herramienta de ci-cd escogida:
    - Jenkins: Jenkinsfile
    - Azure pipelines: azure-pipelines.yml
    - Github actions: .github/workflows/main.yml
    - TravisCI: .travis.yml
    - CircleCI: .circleci/config.yml

- GitlabCI: .gitlab-ci.yml'

8. Se repetía el paso 6 y 7 dos veces más variando la herramienta a utilizar.
9. En este paso se pedía analizar los archivos generados y se daba la posibilidad de hacer preguntas respecto a ellos.
10. Al final se hacía la encuesta de la subsección 5.1.2.8:

5.1.2.7. Resultados esperados

Al realizar esta segunda evaluación se esperan los siguientes resultados:

- Posibles errores y correcciones por realizar en el funcionamiento del generador de pipelines.
- Posibles mejoras al generador de pipelines.
- Percepción de utilidad del generador de pipelines.
- Percepción de compatibilidad del generador de pipelines.
- Percepción de rapidez del generador de pipelines.
- Herramientas similares según la percepción de los desarrolladores.
- Situaciones de posible uso de la herramienta.

5.1.2.8. Encuesta de la evaluación

En la encuesta se definieron 8 preguntas que pudiesen ayudar a responder las preguntas de investigación de esta segunda fase de la subsección 5.1.2.2. A continuación se presentan estas preguntas:

- ¿Cómo calificaría la facilidad de uso de esta herramienta en comparación con sus experiencias previas en la creación de pipelines y despliegue de aplicaciones?
- ¿Qué grado de compatibilidad percibe entre el generador y diversas tecnologías y entornos, basado en su experiencia?
- ¿En comparación con sus métodos anteriores para crear pipelines y desplegar proyectos Java, considera que el generador actual facilita o complica esta tarea? ¿Cómo podría beneficiarle?
- ¿En una escala del 1 al 10, donde 1 indica lento y 10 indica muy rápido, cómo calificaría la eficiencia de este generador frente a los procesos tradicionales?
- ¿Cuáles son las áreas de mejora y temas a corregir que sugiere para la herramienta de generación de pipelines?

- ¿Qué otras herramientas similares conoce y en que se diferencian?
- ¿En qué situaciones usaría la herramienta?
- Teniendo en cuenta que el objetivo es evaluar la herramienta en la satisfacción e intención de uso, ¿Qué pregunta no hice que debí haber hecho?

5.1.2.9. Riesgos de la validez de la evaluación

Al igual que para la fase 1, en la fase 2 hay algunos riesgos a tener en cuenta en el proceso de evaluación. A continuación, se presentan algunos de los riesgos relacionados a la revisión de Zhou et al. (2016) y la forma en que se mitiga cada uno de ellos:

- **Validez de constructo:** La idea de esta categoría es que la evaluación esté encaminada hacia lo que se desea estudiar que en este caso es la utilidad y satisfacción de la herramienta.
  - **Riesgo:** La evaluación podría no estar bien encaminada
    Estrategia de mitigación: La estrategia inicial fue guiarse por el quinto objetivo a cumplir en el presente trabajo de grado. Luego se revisó la definición de *utilidad*[3] y *satisfacción*[4], para así tener claridad de lo que se buscaba encontrar, según este objetivo. Teniendo estos conceptos, se planteó una encuesta que permita responder qué tan útil y satisfechos se sienten las personas con la herramienta.
- **Validez interna:** En este caso se valida qué tan precisa es la evaluación, para ello se identifican algunos riesgos que acarrea el proceso de entrevistas.
  - **Riesgo:** Inconsistencias en el proceso durante las entrevistas
    Estrategia de mitigación: En el proceso de construcción de la entrevista, se creó un formato y guía paso a paso a desarrollar, la cual fue validada por el director y codirector del trabajo de grado. Además, se realizó una prueba previa con un desarrollador, para así validar si el flujo de la entrevista era el adecuado y realizar los ajustes necesarios.
  - **Riesgo:** Los participantes de las entrevistas son todos colombianos y personas conocidas por el investigador.
    Estrategia de mitigación: Los participantes hacen parte de la población objetivo cumpliendo los criterios definidos, adicionalmente, ellos trabajan para diferentes empresas, con diferentes roles y algunas de ellas fuera del país.
  - **Riesgo:** Se tiene un número de participantes pequeño, por lo que puede no ser representativa la muestra.
    Estrategia de mitigación: Para mitigar este riesgo se buscaron participantes que, aunque cumplan los criterios, den variedad a la muestra. Además, se realizaron entrevistas y preguntas que permitan obtener la mayor información posible de los participantes.

[3] https://dle.rae.es/utilidad
[4] https://dle.rae.es/satisfacción

Finalmente, se tiene en cuenta que es posible ampliar la muestra en futuros trabajos y así dar más soporte a la evaluación.

5.2. Resultados de la evaluación

Debido a que se realizaron dos evaluaciones, para la fase 1 y 2, se tienen dos resultados que se presentan a continuación:

5.2.1. Resultados de la fase 1

En esta sección se presentan los resultados obtenidos al evaluar el generador usando proyectos de prueba. Para ello se presentan los datos obtenidos basados en las métricas definidas y así posteriormente responder las preguntas de investigación.

Proyectos / Métricas	**Proyecto Jhipster con Maven**	**Proyecto Jhipster con Gradle**	**InfinityShopping**
Errores de compatibilidad	0	0	0
Errores de generación	5	2	0
Errores en la ejecución	1	1	0
Errores de creación	0	0	0
Errores en el código	1	2	0
Errores en el prompt	0	0	0

Tabla 5.1: Resultados de las métricas definidas - Fase 1

Es importante mencionar que los resultados observados en la tabla 5.1 tienen ciertas consideraciones. Inicialmente, el orden en que se fueron realizando las pruebas fue, primero el proyecto de Jhipster con Maven, luego el proyecto de Jhipster con Gradle y finalmente el proyecto InfinityShopping. En este proceso, cada error que surgió se fue corrigiendo, por tal razón los errores se fueron reduciendo entre los proyectos. Cabe resaltar que todos los pipelines funcionaron correctamente, ya sea desde un principio o luego de hacer las correcciones necesarias.

En el caso de los errores que surgieron, todos fueron debido a errores de codificación, por ejemplo, los errores que se dieron al generar los pipelines fueron en gran medida por temas de identación. Para los temas de errores en el código, por ejemplo, fueron relacionados al uso de los *if* y *else if*. Y finalmente, los errores de ejecución del pipeline se debieron en si al manejo de las credenciales para el despliegue en los diferentes servicios.

A modo de ejemplo, en la figura 5.2 se presenta el proyecto de Jhipster con Gradle desplegado en la App service creada en Azure para el desarrollo de las pruebas.

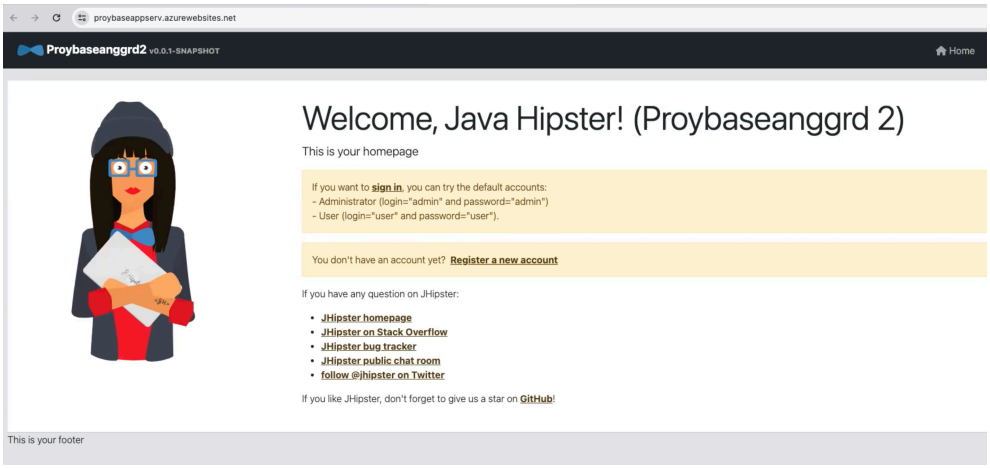


Figura 5.2: Proyecto de Jhipster con Gradle desplegado en Azure

5.2.1.1. Respuesta a las preguntas de investigación de la evaluación

- ¿El generador de pipelines se adapta a las tecnologías presentes en los proyectos de prueba?

  Teniendo en cuenta las pruebas realizadas sobre los proyectos de prueba y la ausencia de errores en cada uno de ellos, se validó que con proyectos de Jhipster que sean generados con una versión superior a 7.3.0 de Jhipster, el generador de pipelines funciona correctamente y no genera ningún problema de compatibilidad con el proyecto generado.

- ¿Es posible generar todas las configuraciones posibles para cada uno de los proyectos sin errores?

  Basados en las pruebas realizadas y la corrección de errores realizada a través de ellos, se validó que es posible realizar las diferentes configuraciones posibles y generar los diferentes pipelines sin problema.

- ¿Los pipelines generados con la herramienta se ejecutan exitosamente en las herramientas de CI-CD?

  Según los resultados obtenidos, la ejecución de los pipelines se realiza exitosamente, sin embargo, si es necesario tener en cuenta las configuraciones adicionales relacionadas a credenciales y permisos de los servicios de Azure sobre los cuales se va a desplegar.

- ¿El generador crea los archivos necesarios para el despliegue en los diferentes servicios?

  En el 100 % de los casos se obtuvo todos los archivos necesarios para la ejecución del pipeline al realizar cada una de las configuraciones posibles.

- ¿El generador crea únicamente el código relacionado a las configuraciones realizadas a través del prompt?

Aunque, se tuvieron algunos errores en el código generado en 3 ocasiones, esto se debió a errores de codificación con las sentencias de condición presentes en el generador, los cuales fueron corregidos en el proceso. Por tal razón al final se obtuvieron cero errores.

- ¿Se presentan únicamente las preguntas y respuestas posibles dependiendo de las respuestas previas en el flujo del prompt?

  El 100 % de las preguntas y posibles respuestas presentes en el prompt del generador de pipelines fueron los correspondientes para cada paso, permitiendo llevar un flujo adecuado para lograr generar los pipelines exitosamente

5.2.2. Resultados de la fase 2

En esta sección se presentan los resultados obtenidos mediante el proceso de entrevistas a desarrolladores evaluando el generador. Durante la evaluación y con el objetivo de cuantificar las respuestas, se preguntó a los desarrolladores en el caso de la utilidad, compatibilidad y rapidez que dieran un valor entre 1 y 10 que se relacione con su percepción. Para ello se presentan los datos obtenidos basados en las métricas definidas y así posteriormente responder las preguntas de investigación.

Participantes / Métricas	**P1**	**P2**	**P3**	**P4**	**P5**	**P6**	**P7**	**P8**
Percepción de utilidad	7	10	8	9	8	10	8	8
Percepción de compatibilidad	10	10	10	5	7	7	7	10
Percepción de rapidez	10	10	10	9	10	6	10	10
Número de mejoras	3	2	2	2	2	3	3	7
Número de correcciones	0	0	0	0	0	0	0	0
Número de herramientas similares	1	1	1	1	1	0	0	1

Tabla 5.2: Resultados de las métricas definidas - Fase 2

5.2.2.1. Respuesta a las preguntas de investigación de la evaluación

- *¿Qué tan fácil es usar la herramienta en comparación con otras herramientas para crear pipelines?*

  Al validar la utilidad del generador de pipelines con los desarrolladores, se obtuvo un valor promedio de 8,625, siendo el 10 el valor máximo, según las calificaciones dadas. El cual es un valor que permite intuir que la herramienta es bastante útil para los desarrolladores. Además, durante las entrevistas se dieron múltiples comentarios indicando la utilidad de la herramienta, ya que, les facilitaba el trabajo en gran medida al realizar despliegues, la reducción de tiempos y la mitigación de errores humanos en la creación de los pipelines.

Cabe resaltar que una de las razones por las cuales este valor bajó un poco, fue debido a la interfaz de usuario que se utiliza en la herramienta, la cual es la consola de comandos. Ya que, preferían una interfaz más amigable y que genere cierta confianza en su uso.

- *¿Qué grado de compatibilidad hay entre el generador y diversas tecnologías del ambiente laboral?*

  Al calificar la compatibilidad de la herramienta, los desarrolladores dieron un valor promedio de 8,25. Esto nos da una buena calificación de percepción con relación a la compatibilidad, teniendo en cuenta la variedad de tecnologías que articula el generador, el cual fue uno de los comentarios de los participantes, ya que, la herramienta les permitía generar pipelines para algunas de las herramientas más populares en el mercado.

  La razón que redujo este valor se debió al acoplamiento del generador a Jhipster, lo cual no permite generar pipelines para proyectos que no se hayan generado con Jhipster o que tengan su estructura. Adicionalmente, se mencionó la falta de AWS como nube para los servicios, por lo que algunos de ellos han centrado su experiencia con esta nube.

- *¿El generador facilita el desplegar proyectos Java que se compilen con Gradle y o Maven?*

  La validación de esta pregunta fue de forma cualitativa, con lo cual se obtuvieron comentarios afirmativos, ya que el tener los dos tipos de compilación, se cubre en gran medida las opciones disponibles en el mercado. Sin embargo, se sugirió que la herramienta no solo se enfoque en Java, ya que los demás lenguajes como Python y .Net han extendido su uso y por tanto sería muy útil tenerlos disponibles.

- *¿Qué tan rápido es el generador, en comparación a procesos tradicionales?*

  Basados en los valores dados por los desarrolladores, se obtuvo un promedio de 9,375. Lo cual, confirma que la herramienta brinda una gran rapidez al generar los pipelines, ya que como lo mencionaban, en muy poco tiempo obtenían una plantilla funcional de un pipeline. Cabe mencionar que uno de los valores dados fue bajo, debido a que para la percepción del desarrollador la ejecución del pipeline tomaba mucho tiempo, sin embargo, esto se debe a las pruebas y configuraciones agregadas al pipeline y que Jhipster también tiene.

- *¿Cuáles son las áreas de mejora y temas a corregir para la herramienta de generación de pipelines?*

  Teniendo en cuenta el número de mejoras sugeridas por los desarrolladores, se obtuvieron un total de 24. Sin embargo, algunas eran iguales o similares, por tanto, a continuación se presentan estas mejoras condensadas:

  1. Independizar el generador de Jhipster, una de las opciones es preguntar al inicio si se desea el pipeline para un proyecto Jhipster o para uno que no lo sea, así no se pierde la versión actual y también se brinda la opción fuera de Jhipster.
  2. Crear una funcionalidad adjunta a la herramienta que permita analizar su uso y permitir a los usuarios brindar sugerencias y así con los dos datos, enfocar las posibles mejoras.

3. Crear una interfaz más amigable al usuario, en lugar de utilizar la consola de comandos. Lo cual sugieren es más cómodo y confiable. Un ejemplo dado es Spring Initializr.
4. Crear un manual de usuario que este adjunto a la herramienta y se pueda acceder de forma fácil, lo cual explique cómo usarla y qué configuraciones y requisitos adicionales se deben hacer para su funcionamiento y para el funcionamiento de los pipelines. Adicionalmente, relacionar la documentación de las otras herramientas que usa o se relacionan con el generador.
5. Crear un manual técnico que permita a otros desarrolladores entender el funcionamiento de la herramienta y así implementar sus mejoras.
6. Ordenar las variables de una forma más amigable, ubicándolas al inicio de los pipelines para una mejor visualización.
7. Implementar un módulo en el generador o un generador adicional que permita crear Infraestructura como código (IaC) y así crear los recursos con la misma facilidad que se crean los pipelines.
8. No usar los scripts definidos en el package.json y en lugar de este se sugiere utilizar Make para simplificar cada uno de los pasos en los pipelines. Esta es una posible opción para desligar la herramienta de Jhipster.
9. Agregar a los pipelines una opción para enviar una notificación al usuario cuando finalice su ejecución o surga un posible error.
10. Incluir otras opciones de nube para desplegar, dando prioridad a AWS, ya que basados en los comentarios del desarrollador es la más usada.
11. Implementar una opción para convertir pipelines entre las diferentes herramientas de CI/CD y así apoyar procesos de migración.
12. Agregar comentarios a los pipelines que expliquen las diferentes secciones de este y que hace cada una. De esa manera personas que no tengan muchos conocimientos en pipelines puedan entenderlos de una mejor manera.
13. Al generar el pipeline no tomar la rama master por defecto para el disparador, en su lugar se debería tomar la rama activa en el proyecto.

- *¿Qué otras herramientas similares hay y en que se diferencian?*

  A continuación, se mencionan las herramientas mencionadas por los desarrolladores, que basados en su percepción, se asemejan al generador de pipelines.

  1. **Chatgpt:** ChatGPT es un modelo de lenguaje desarrollado por OpenAI que utiliza inteligencia artificial para generar texto en lenguaje natural. Sin embargo, ChatGPT no es una herramienta de desarrollo o de generación de pipelines. Es una herramienta de IA que puede ayudar con consultas relacionadas con programación, entre otros temas, pero no está enfocada en generar pipelines de CI/CD, ni se integra directamente con herramientas de desarrollo

2. **Bito:** Bito es una herramienta que utiliza inteligencia artificial para ayudar a los desarrolladores con sus tareas, proporcionando sugerencias de código, completando código automáticamente, y mejorando la productividad. Este se enfoca en mejorar la productividad del desarrollador dentro del editor de código, mas no está diseñado específicamente para generar pipelines de CI/CD como el generador de pipelines.
3. **Azure pipelines:** Azure Pipelines es una parte de Azure DevOps que proporciona servicios de CI/CD para la construcción, prueba y despliegue automático de aplicaciones. Esta contiene una funcionalidad *drag and drop*, que crea código para sus pipelines, según lo que se seleccione. Sin embargo, Azure pipelines es solo una de las herramientas de CI/CD, en cambio con el generador es posible crear pipelines para varias herramientas de CI/CD.
4. **Github Copilot:** GitHub Copilot es un asistente de codificación AI desarrollada por GitHub y OpenAI. Sugiere líneas de código o funciones completas mientras el desarrollador escribe. GitHub Copilot ayuda a escribir código proporcionando sugerencias inteligentes, más no genera pipelines de CI/CD directamente.
5. **Yeoman:** Yeoman es una herramienta de scaffolding para aplicaciones web. Ofrece generadores que permiten a los desarrolladores configurar rápidamente la estructura de un proyecto. Aunque Yeoman se puede usar como generador de código en general, este enfoca sus desarrollos en aplicaciones web, lo cual, se diferencia con el enfoque del generador de pipelines.
6. **Spring Inicializr:** Spring Initializr es una herramienta que permite a los desarrolladores generar rápidamente un proyecto Spring Boot inicial con dependencias preconfiguradas. Son similares en su idea de generar código y estructuras base el desarrollo de software, pero su diferencia reside en el enfoque de la herramienta, No se enfoca en la generación de pipelines de CI/CD.

- *¿En qué situaciones los desarrolladores usarían la herramienta?*

  En esta última pregunta, la mayoría de los desarrolladores mencionaron que el uso que le darían a la herramienta sería de forma personal, para prácticas, emprendimientos o en proyectos fuera de su entorno laboral debido a la limitación por la dependencia de Jhipster. Sin embargo, un par de desarrolladores si mencionaron la intención de usar la herramienta en su entorno laboral, pero con bastante cuidado, validando cada configuración que la herramienta les pudiese brindar.

5.3. Resumen del Capítulo

La evaluación de nuestra propuesta para la generación de pipelines de CI/CD se realizó en dos fases: Una primera fase donde se evaluó la herramienta mediante una serie de proyectos de prueba, y una segunda fase mediante la revisión con un grupo de ocho (8) desarrolladores de software.

Este capítulo describe la evaluación en dos secciones: La primera sección describe el diseño de las dos fases de evaluación, presenta las fases de la evaluación, las preguntas de investigación usadas en la evaluación, la muestra seleccionada, las métricas para medir la evaluación, los resultados esperados y finalmente los riesgos de cada evaluación. La segunda sección del capítulo presenta los resultados de la evaluación, los datos obtenidos en cada revisión y las respuestas a las preguntas planteadas.

En la primera fase de evaluación, enfocada en validar el funcionamiento de la herramienta, se percibieron algunos errores de generación, de código y de ejecución. Esta evaluación permitió mejorar la herramienta, solventando errores y brindando, al final, un generador que funciona sin errores en todos los proyectos de prueba.

La segunda fase 2 de la evaluación, enfocada en la revisión con desarrolladores, permitió confirmar la utilidad de la herramienta, la intención de uso por parte de los desarrolladores y la satisfacción con respecto a la compatibilidad y rapidez. Además, esta evaluación permitió identificar puntos de mejora que permitirían potenciar la herramienta en el futuro.

Capítulo 6

Conclusiones, lecciones aprendidas y trabajos futuros

El presente trabajo de grado tenía como objetivo brindar una herramienta para la generación de pipelines de CI/CD utilizando técnicas de reutilización y variabilidad. Una herramienta que permita a los desarrolladores ahorrar tiempo y reducir los errores que se dan al crear pipelines manualmente.

El presente capítulo presenta las conclusiones del desarrollo de esta herramienta, dando respuesta a los objetivos planteados, basándose en la información obtenida en todo el proyecto. Adicionalmente, se presentan lecciones aprendidas y posibles trabajos futuros relacionados al presente trabajo de grado.

6.1. Conclusiones

Teniendo en cuenta los objetivos planteados para la creación de una herramienta de generación de pipelines para CI/CD se obtuvieron las siguientes conclusiones:

Para dar solución al primer objetivo de este trabajo, se realizó un análisis del estado del arte, donde se identificaron varios proyectos y herramientas similares al generador de pipelines propuesto. A pesar de que muchas de estas herramientas presentaban ciertas falencias, respecto al objetivo general del presente trabajo de grado, su estudio permitió extraer y analizar técnicas útiles para la creación de nuestra herramienta, como se detalla en el capítulo 3.

Además, en el capítulo 2 se desarrolló un marco teórico que proporcionó una base de conocimiento esencial sobre las diferentes técnicas de reutilización y variabilidad disponibles. Este marco fue fundamental para entender las posibilidades y limitaciones de cada técnica, permitiendo una selección informada.

Con la información obtenida, se decidió adoptar la técnica de línea de productos de software en el capítulo 4. Esta técnica permitió integrar de manera efectiva los conceptos de reutilización y variabilidad, cumpliendo así con el objetivo de identificar y definir una metodología que facilite la reutilización y variación del código de pipelines para CI/CD.

Para alcanzar el segundo objetivo de este proyecto, en el capítulo 4, se establecieron y utilizaron varios atributos de calidad, como compatibilidad, usabilidad, funcionalidad, entre otros, los cuales guiaron efectivamente el desarrollo del generador de pipelines. La selección de herramientas tecnológicas adecuadas y la implementación de la técnica de línea de productos de software fueron fundamentales para el éxito del proyecto.

El análisis de dominio permitió definir un modelo de características exhaustivo, lo cual facilitó la identificación de las distintas funcionalidades que debería incluir el generador de pipelines. A través del análisis de requerimientos, se detallaron los requisitos específicos y las características esenciales para los productos a generar, asegurando que el desarrollo fuera alineado con las necesidades reales del usuario.

La implementación del dominio, incluyendo la estructura del generador y la definición de la variabilidad, permitió crear una herramienta flexible y adaptable. Este enfoque permitió que la variabilidad se gestionara de manera eficiente en distintos momentos del ciclo de vida del software, utilizando tecnologías apropiadas y enfoques específicos como anotaciones.

El desarrollo final de la herramienta, con sus prompts y templates configurables, demostró ser robusto y versátil, capaz de generar diferentes pipelines según las necesidades específicas. Los ejemplos de pipelines generados confirmaron la eficacia y la aplicabilidad de la herramienta en entornos reales.

Luego para lograr el tercer objetivo específico, se realizó una evaluación exhaustiva del generador de pipelines a través de proyectos de prueba, con el propósito de validar su correcto funcionamiento. Durante la fase inicial de esta evaluación, se detectaron varios errores en la generación, código y ejecución de los pipelines. Sin embargo, este proceso fue crucial para identificar y corregir dichos errores, mejorando significativamente la herramienta. Al finalizar esta fase, el último proyecto de prueba no presentó errores, demostrando así la robustez y eficacia del generador de pipelines. Las respuestas a las preguntas de investigación y los datos obtenidos de las métricas definidas confirmaron que la herramienta funciona correctamente, cumpliendo así el tercer objetivo específico del proyecto.

Finalmente, Para llevar a cabo el cuarto objetivo específico y evaluar la satisfacción y utilidad del generador de pipelines, se llevó a cabo una segunda fase de evaluación con la participación de 8 desarrolladores de software que utilizan pipelines en su labor diaria. Los resultados de esta fase indicaron una alta satisfacción en términos de compatibilidad y rapidez de la herramienta. Los desarrolladores confirmaron la utilidad del generador y mostraron una clara intención de uso en sus proyectos futuros. Además, se recopilaron valiosos comentarios de mejora, que proporcionan una base sólida para futuras optimizaciones y aumentarán aún más la satisfacción y el valor percibido de la herramienta. Estos resultados positivos validan la propuesta y demuestran su potencial impacto en la mejora de procesos de CI/CD.

6.2. Lecciones aprendidas

Durante el desarrollo de este trabajo de grado, he adquirido conocimientos valiosos y experiencia práctica en varias áreas clave del desarrollo de software, especialmente en lo que respecta a la integración, entrega y despliegue continuo. A continuación, se detallan las lecciones aprendidas a lo largo de este proceso.

Inicialmente puede aprender sobre los procesos de integración continua (CI), entrega continua (CD) y despliegue continuo, así como la importancia de los pipelines en el desarrollo de software moderno. Exploré y utilicé diversas herramientas como Azure Pipelines, GitHub Actions, Jenkins,

Travis CI y CircleCI. Cada una de estas herramientas tiene sus propias características, lenguajes de configuración y ventajas, y su correcta implementación puede aumentar significativamente la eficiencia y la calidad del desarrollo de software en contraste con métodos tradicionales.

Además, aprendí sobre los diferentes servicios y configuraciones necesarias para desplegar aplicaciones web utilizando Azure Pipelines. Entender las consideraciones específicas y los ajustes necesarios para un despliegue exitoso desde un pipeline fue fundamental para asegurar la automatización y eficiencia del proceso de despliegue.

Al usar una herramienta como JHipster, pude aprender sobre él y su capacidad para generar código de calidad que facilita el desarrollo rápido de aplicaciones. La experiencia de trabajar con un proyecto de código abierto de la comunidad no solo mejoró mis habilidades técnicas, sino que también me permitió contribuir de vuelta a la comunidad, lo cual es una experiencia enriquecedora.

Por otra parte, a pesar de tener conocimientos previos sobre las líneas de productos de software, este proyecto me permitió profundizar y consolidar mi comprensión. Implementar una línea de productos de software y seguir sus pasos me proporcionó una metodología sólida que podré aplicar en futuros desarrollos, mejorando la reutilización y variabilidad del código en mis proyectos.

Y finalmente, la interacción con los desarrolladores durante la fase de evaluación me permitió descubrir nuevas herramientas y perspectivas que puedo aplicar en futuros proyectos. Los comentarios y sugerencias recibidos no solo ayudaron a mejorar la herramienta desarrollada, sino que también me brindaron una visión más amplia de las necesidades y expectativas de los usuarios finales.

En resumen, este trabajo de grado ha sido una experiencia de aprendizaje integral que ha mejorado mis habilidades técnicas y de colaboración, proporcionándome conocimientos prácticos y teóricos que serán fundamentales para mi carrera futura en el desarrollo de software.

6.3. Trabajos futuros

El desarrollo del generador de pipelines para CI/CD ha abierto diversas oportunidades para futuras mejoras y expansiones. A continuación, se detallan algunas de las áreas clave para trabajos futuros que podrían potenciar y ampliar significativamente la utilidad y alcance de la herramienta.

Una primera línea de trabajo consiste en desacoplar el generador de pipelines de JHipster, permitiendo así que funcione de manera independiente. Esto permitiría generar pipelines para cualquier tipo de proyecto, no solo aquellos creados con JHipster, ampliando considerablemente su aplicación y usabilidad en diferentes entornos de desarrollo.

Además, es esencial trabajar en la automatización y adaptación de la lectura de los archivos de configuración presentes en los proyectos. Al hacerlo, el generador podría crear pipelines de manera más automática y mejor adaptada a cada proyecto específico, sin perder la flexibilidad para que los usuarios puedan personalizar las configuraciones según sus necesidades particulares.

Otra mejora significativa sería trasladar el generador de pipelines desde la consola de comandos hacia una interfaz gráfica de usuario (GUI) más amigable. Una GUI intuitiva no solo facilitaría el uso de la herramienta para los desarrolladores menos familiarizados con la línea de comandos,

sino que también mejoraría la experiencia del usuario al hacer más accesibles y comprensibles las funcionalidades del generador.

Incrementar la variabilidad del generador de pipelines es otro ámbito prometedor para trabajos futuros. Integrar soporte para otras plataformas de nube como AWS y Google Cloud, además de los diversos servicios que estas ofrecen, podría atraer a un espectro más amplio de usuarios. Además, extender la capacidad de generar pipelines para distintos tipos de proyectos, incluyendo microservicios, lambdas, e incluso aplicaciones móviles, podría aumentar la flexibilidad y utilidad de la herramienta.

También se podría mejorar el generador de pipelines actual mediante la integración de funcionalidades adicionales, como notificaciones automáticas a los usuarios cuando los pipelines se completen u ocurran errores. Utilizar servicios de notificación como correo electrónico o mensajes podría mantener a los desarrolladores informados en tiempo real. Además, mejorar las descripciones dentro de los pipelines y acompañarlas de guías detalladas ayudaría a los usuarios a comprender mejor el proceso y las configuraciones necesarias.

La capacidad de generar pipelines para proyectos específicos utilizando diversas herramientas de CI/CD abre la posibilidad de realizar un benchmarking comparativo. Esto permitiría evaluar y comparar la eficiencia y rendimiento de las diferentes herramientas integradas, proporcionando datos valiosos para la toma de decisiones sobre qué herramientas utilizar en distintos escenarios.

Finalmente, una expansión interesante sería el desarrollo de más herramientas para generar otros tipos de códigos, como Infraestructura como Código (IaC). Utilizar herramientas como Terraform o CloudFormation para este propósito no solo ampliaría la funcionalidad del generador, sino que también proporcionaría un mayor nivel de reutilización y variabilidad, beneficiando a los desarrolladores que buscan soluciones completas y adaptables.

En resumen, las futuras mejoras y expansiones del generador de pipelines tienen el potencial de transformar esta herramienta en una solución aún más robusta y versátil, beneficiando a una comunidad más amplia de desarrolladores y adaptándose a una variedad de entornos y necesidades de proyectos.

Bibliografía

Apel, S., Batory, D., Kästner, C., and Saake, G. (2013). *Feature-Oriented Software Product Lines*. Springer Berlin, Heidelberg.

Aydin, S., Steffens, A., and Lichter, H. (2021). Automated construction of continuous delivery pipelines from architecture models. In *28th Asia-Pacific Software Engineering Conference, APSEC 2021, Taipei, Taiwan, December 6-9, 2021*, pages 306–316. IEEE.

Baehr, M. (2004). Evaluation methodology. *Pacific Crest, Faculty Development Series*.

Beck, K. (1999). *Extreme Programming Explained: Embrace Change*. Addison-Wesley Publishing Company.

Booch, G. (1990). *Object Oriented Design with Applications*. Benjamin-Cummings Publishing Co., Inc., USA.

Davidson, J. (2005). *Evaluation Methodology Basics: The Nuts and Bolts of Sound Evaluation*. SAGE Publications.

Debroy, V., Miller, S., and Brimble, L. (2018). Building lean continuous integration and delivery pipelines by applying devops principles: A case study at varidesk. *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*.

Djounang-Nana, G. R. (2022). Towards the automatic identification of optimal configurations of ci/cd pipelines of software development projects: Symbolic, meta-heuristic and statistical approaches. *The 20th International Conference on Software and Systems Reuse (ICSR-2022)*.

Donca, I.-C., Stan, O. P., Misaros, M., Gota, D., and Miclea, L. (2022). Method for continuous integration and deployment using a pipeline generator for agile software projects. *Sensors*, 22(12).

Duvall, P., Matyas, S., and Glover, A. (2007). *Continuous Integration: Improving Software Quality and Reducing Risk*. Addison-Wesley Professional, first edition.

Gamma, E., Helm, R., Johnson, R., and Vlissides, J. (1994). *Design Patterns, Elements of Reusable Object-Oriented Software*. Pearson Education.

Golzadeh, M., Decan, A., and Mens, T. (2021). On the rise and fall of ci services in github. In *9th IEEE International Conference onSoftware Analysis, Evolution and Reengineering (SANER 2022)*. IEEE.

Humble, J. and Farley, D. (2010). *Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation*. Addison-Wesley Professional, 1st edition.

Ivanov, V. and Smolander, K. (2018). Implementation of a devops pipeline for serverless applications. *Lecture Notes in Computer Science*, 11271 LNCS:48–64.

Jones, C. (2018). Using code generation to enforce uniformity in software delivery pipelines. In Bruel, J., Mazzara, M., and Meyer, B., editors, *Software Engineering Aspects of Continuous Development and New Paradigms of Software Production and Deployment - First International Workshop, DEVOPS 2018, Chateau de Villebrumier, France, March 5-6, 2018, Revised Selected Papers*, volume 11350 of *Lecture Notes in Computer Science*, pages 155–168. Springer.

Jumani, A. K., Shaikh, A. A., Khan, M. O., and Siddique, W. A. (2020). Fast delivery, continuously build, testing and deployment with devops pipeline techniques on cloud. *Indian Journal of Science and Technology*, 13:552–575.

Kitchenham, B. A. and Charters, S. (2007). Guidelines for performing systematic literature reviews in software engineering. Technical Report EBSE 2007-001, Keele University and Durham University Joint Report.

Laaja, O. (2022). Design of an automated pipeline to improve the process of cross-platform mobile building and deployment. Tesis de maestría, University of Helsinki.

Labouardy, M. (2021). *Pipeline as Code*. Manning Publications Co.

Lachhman, R. (2022). *Pipeline Patterns for CI/CD*. Harness Inc.

Padhye, P., Khawale, A., and Professor, G. A. A. (2023). Streamlining software development: An approach to ci/cd pipeline automation. *Vidhyayana E-Journal*, 8.

Poppendieck, M. and Poppendieck, T. (2003). *Lean Software Development: An Agile Toolkit*. Addison-Wesley Longman Publishing Co., Inc., USA.

Priyadarsini, K., Fantin, E., Raj, I., Begum, A. Y., and Shanmugasundaram, V. (2020). Comparing devops procedures from the context of a systems engineer. *Materials Today: Proceedings*.

Renault, O. (2014). Reuse / variability management and system engineering. In *Poster Workshop at the 2014 Complex Systems Design and Management International Conference (CSDM 2014 Poster Workshop)*, CEUR Workshop Proceedings.

Sommerville, I. (2016). *Software engineering*. Pearson Education, 10th edition.

Tebes, G., Peppino, D., Rivera, B., Becker, P., Papa, F., and Olsina, L. (2019). Especificación del proceso de design science research: Caso aplicado a una ontología de testing de software. In *7mo Congreso Nacional de Ingeniería Informática/Sistemas de Información (CoNaIISI)*.

TechTarget (2021). E-guide ci/cd pipelines explained: Everything you need to know. Technical report, TechTarget.

van Gurp, J., Bosch, J., and Svahnberg, M. (2002). On the notion of variability in software product lines. Tesis de licenciatura en ingeniería, Blekinge Institute of Technology.

vom Brocke, J., Hevner, A., and Maedche, A. (2020). Introduction to design science research. In *Design Science Research*, pages 1–13. Springer.

Zhou, X., Jin, Y., Zhang, H., Li, S., and Huang, X. (2016). A map of threats to validity of systematic literature reviews in software engineering. In *2016 23rd Asia-Pacific Software Engineering Conference (APSEC)*, pages 153–160.