

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE INFORMÁTICA
SANTIAGO - CHILE



“TRANSFORMACIÓN EN LA ENTREGA Y DESARROLLO
DE SOFTWARE: USO DE AUTOMATIZACIÓN PARA
MEJORAR LA EFICIENCIA Y CALIDAD EN EL
DESARROLLO”

MATÍAS ALBERTO CONTRERAS AID

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN INFORMÁTICA

Profesor Guía: Marcello Visconti Zamora
Profesor Correferente: Ricardo Salas

Marzo - 2026



CONSTANCIA DE VALIDACIÓN Y CONFIDENCIALIDAD DE MONOGRAFÍA A REPOSITORIO ACADÉMICO

1.- IDENTIFICACIÓN DEL TRABAJO ACADÉMICO

Tipo de monografía (marcar una opción): ☒ Memoria o trabajo de título ☐ Tesis de postgrado

Titulo del trabajo: Transformación en la entrega y desarrollo de software: Uso de automatización para mejorar la eficiencia y calidad en el desarrollo

Nombre del candidato(a): Matías Alberto Contreras Aid

Carrera/Grado: Ingeniería Civil Informática / Pregrado

Campus: San Joaquín

Departamento: Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Marcello Visconti Zamora, en mi calidad de profesor(a) guía/director(a) del trabajo académico mencionado anteriormente **DEJO CONSTANCIA** que:

- He revisado esta versión del documento y corresponde a la versión final aprobada del trabajo.
- El trabajo cumple con los requisitos académicos y de formato establecidos por la institución.

3. EVALUACIÓN DE CONFIDENCIALIDAD POR PROPIEDAD INDUSTRIAL (marcar una opción)

☒ El trabajo **NO contiene** información que amerite confidencialidad y puede ser publicado de inmediato en el repositorio con acceso abierto.

☐ El trabajo **CONTIENE** información con potenciales implicancias de propiedad industrial o intelectual y requiere un periodo de confidencialidad (**embargo**) por (**marcar opción**):

☐ 6 meses ☐ 12 meses ☐ 2 años ☐ 3 años ☐ 5 años ☐ 10 años

Fundamentación de la necesidad de confidencialidad (obligatorio si se solicita embargo):

4. FIRMAS

Profesor(a) guía o director(a) de memoria o tesis:

Fecha: 20/03/2026

Firma:

Estudiante o Candidato(a):

Fecha: 20/03/2026

Firma:

DEDICATORIA

A Javiera, mi pareja, por siempre estar a mi lado en todos los momentos, apoyándome y motivándome a seguir adelante.

A mis padres, mi hermano y mi tía por su amor incondicional y su apoyo constante.

A mis amigos de carrera, Vicente Lineros, Juanfra y Vicente Figueroa por su compañía, los buenos ratos y por hacer que este camino sea más llevadero.

A mis amigos de toda la vida, Isra, Vicente, Vicho, Pablo por su apoyo a lo largo de toda mi vida.

RESUMEN

Resumen— En equipos pequeños de desarrollo de software es habitual la adopción parcial de prácticas de ingeniería y automatización, lo que incrementa el esfuerzo manual, el riesgo operacional y la variabilidad en los tiempos de entrega. Este trabajo caracteriza dichas brechas mediante un diagnóstico basado en una muestra acotada de proyectos reales, contrastando las prácticas observadas con recomendaciones de la literatura e industria en torno a estrategia de ramificación en Git, integración continua, entrega/despliegue continuos e Infraestructura/Configuración como Código. El análisis evidencia patrones recurrentes de baja madurez: ausencia de CI, despliegues manuales u opacos dependientes de terceros, IaC y limitada trazabilidad entre cambios, artefactos y despliegues. A partir de estas brechas se diseña un proceso incremental, estructurado en fases con prácticas mínimas, criterios de aceptación y artefactos esperados, orientado a reducir trabajo manual, mejorar reproducibilidad de entornos y asegurar trazabilidad explícita por ambiente. Finalmente, se define un esquema de medición para comparar el desempeño antes y después de la adopción del proceso, utilizando métricas operacionales tales como *lead time*, frecuencia de despliegue, tasa de fallos por cambio y tiempo medio de recuperación.

Palabras Clave— DevOps, Automatización, Calidad, Trazabilidad, Equipos pequeños

ABSTRACT

Abstract— Small software teams often adopt engineering and automation practices only partially, increasing manual effort, operational risk, and variability in delivery times. This work characterizes such gaps through a diagnosis based on a bounded sample of real projects, contrasting observed practices with industry and academic recommendations regarding Git branching strategies, continuous integration, continuous delivery/deployment, and Infrastructure/Configuration as Code. The diagnosis reveals recurring low-maturity patterns: lack of CI, manual or opaque deployments dependent on third parties, low or no use of IaC, and limited traceability between changes, artifacts, and deployments. Based on these gaps, an incremental process is designed and organized into phases with minimum practices, acceptance criteria, and expected artifacts, aiming to reduce manual work, improve environment reproducibility, and ensure explicit per-environment traceability. Finally, a measurement scheme is defined to compare performance before and after process adoption, using operational metrics such as lead time, deployment frequency,

change failure rate, and mean time to recovery.

Keywords— DevOps, Automation, Quality, Traceability, Small Teams

GLOSARIO

Aquí se deben colocar las siglas mencionadas en el trabajo y su explicación, por orden alfabético. Por ejemplo:

AWS: *Amazon Web Services*. Plataforma de servicios en la nube de Amazon.

Azure: Plataforma de servicios en la nube de Microsoft.

Bitbucket: Plataforma de alojamiento de repositorios Git y herramientas de colaboración (Atlassian).

CaC: *Configuration as Code*. Práctica de definir y versionar configuraciones (sistemas, servicios o aplicaciones) como artefactos tratables como código.

CD: *Continuous Delivery/Continuous Deployment*. Conjunto de prácticas y automatizaciones para mantener el software siempre desplegable y/o desplegar cambios de forma frecuente y confiable.

CI: *Continuous Integration*. Práctica de integrar cambios con alta frecuencia en un repositorio compartido y ejecutar automáticamente construcción y validaciones (p. ej., pruebas) para detectar fallos temprano.

CFR: *Change Failure Rate*. Métrica DORA que estima la proporción de cambios/despliegues que resultan en fallos (p. ej., incidentes o rollbacks).

DevOps: Enfoque cultural y técnico que integra desarrollo y operación para mejorar colaboración, automatización y retroalimentación a lo largo del ciclo de vida del software.

DSW: Desarrollo de Software.

FD: *Deployment Frequency*. Métrica DORA que representa la frecuencia con la que un equipo despliega cambios a un ambiente objetivo (por ejemplo, producción).

Git: Sistema de control de versiones distribuido utilizado para registrar historial de cambios y facilitar trabajo colaborativo.

GitHub: Plataforma de alojamiento de repositorios Git y colaboración (por ejemplo, *issues*, *pull requests* y automatizaciones).

Gitflow: Estrategia de ramificación basada en ramas largas (por ejemplo, *develop* y *main*) y ramas de soporte (feature/release/hotfix).

GitHub Actions: Servicio de automatización en GitHub que permite ejecutar workflows (pipelines) ante eventos del repositorio.

GitLab: Plataforma para repositorios y automatización CI/CD.

GCP: *Google Cloud Platform*. Plataforma de servicios en la nube de Google.

IaC: *Infrastructure as Code*. Práctica de aprovisionar infraestructura mediante definiciones versionadas (declarativas o imperativas) en lugar de configuraciones manuales.

IaaS: *Infrastructure as a Service*. Modelo cloud donde se proveen recursos de cómputo, redes y almacenamiento como servicio.

IoT: *Internet of Things*. Paradigma donde dispositivos físicos conectados a Internet intercambian datos y eventos.

LT: *Lead Time*. Métrica DORA que representa el tiempo desde un cambio de código (o decisión equivalente) hasta su disponibilidad en un ambiente objetivo.

MR: *Merge Request*. Solicitud de integración de cambios, comúnmente usada en GitLab.

MTTR: *Mean Time to Restore*. Métrica DORA que representa el tiempo promedio para restaurar el servicio tras un incidente.

Pipeline: Secuencia automatizada de etapas (por ejemplo, build, test, despliegue) ejecutadas por una herramienta CI/CD.

PR: *Pull Request*. Solicitud de cambio para revisión e integración de código (comúnmente usada en GitHub).

Producción: Ambiente operacional donde el sistema presta servicio a usuarios finales.

QA: *Quality Assurance*. Función o rol asociado a asegurar calidad, incluyendo pruebas y control de procesos.

Rollback: Mecanismo para revertir un despliegue o cambio a una versión previa estable.

SaaS: *Software as a Service*. Modelo de entrega de software como servicio accesible por red.

SHA: *Secure Hash Algorithm*. En contexto Git, identificador hash (por ejemplo, SHA-1/SHA-256) que referencia un commit.

SQS: *Simple Queue Service*. Servicio de colas administrado (AWS) para desacoplar componentes mediante mensajería.

TDD: *Test-Driven Development*. Técnica en la que se escriben pruebas antes de implementar funcionalidad, siguiendo el ciclo Red-Green-Refactor.

Terraform: Herramienta de IaC orientada a definir y aprovisionar recursos mediante configuraciones declarativas.

ÍNDICE DE CONTENIDOS

RESUMEN	III
ABSTRACT	III
GLOSARIO	V
ÍNDICE DE FIGURAS	IX
ÍNDICE DE TABLAS	IX
INTRODUCCIÓN	1
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA	2
1.1 Contexto	2
1.2 Diagnóstico y análisis de brecha (gap analysis)	3
1.2.1 Alcance del diagnóstico	3
1.2.2 Recolección de información real	3
1.2.3 Análisis de brecha	8
1.2.4 Priorización de problema	12
1.3 Causas y efectos del problema	12
1.3.1 Problema central	12
1.3.2 Causas	13
1.3.3 Efectos	14
1.4 Prácticas adoptables	14
1.5 Alcances y limitaciones	15
1.6 Objetivos	15
1.6.1 Objetivo general	15
1.6.2 Objetivos específicos	15
CAPÍTULO 2: MARCO CONCEPTUAL	16
2.1 Metodologías Ágiles	16
2.1.1 Desarrollo dirigido por tests (TDD)	16
2.2 DevOps	17
2.2.1 Integración continua (CI)	17
2.2.2 Entrega y despliegue continuos (CD)	18
2.2.3 Infraestructura como código (IaC)	19
2.2.4 Configuración como código (CaC)	19
2.3 Control de versiones con Git	19
2.3.1 Estrategias de ramificación	20
2.4 Infraestructura Cloud	20
CAPÍTULO 3: PROPUESTA DE SOLUCIÓN	22
3.1 Definiciones operativas y métricas de evaluación	22

3.2	Alcance, supuestos y principios de diseño	23
3.3	Estructura general del proceso	23
3.4	Trazabilidad entre brechas del diagnóstico y el proceso	24
3.5	Descripción de las fases del proceso	25
3.5.1	Fase 1: Preparación y establecimiento de línea base	25
3.5.2	Fase 2: Control de cambios y calidad mínima	26
3.5.3	Fase 3: Empaquetado y despliegue continuo (CD) con verificaciones	27
3.5.4	Fase 4: IaC/CaC y reproducibilidad de entornos	28
3.5.5	Fase 5: Operación, trazabilidad y reversión	28
3.5.6	Fase 6: Medición, análisis y ajuste	29
3.6	Estrategia de adopción y artefactos de apoyo	30
3.7	Trazabilidad con la validación	30
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN		31
4.1	Diseño del estudio	31
4.1.1	Ventanas de observación y comparabilidad	31
4.2	Caracterización de los casos	32
4.3	Alcance de aplicación del proceso	33
4.3.1	Definiciones operativas adicionales: <i>runbook</i> y <i>rollback drill</i>	34
4.4	Métricas, fuentes de evidencia y método de cálculo	35
4.5	Presentación de evidencia y trazas	35
4.5.1	Muestra de despliegues y tiempos observados	36
4.6	Resultados cuantitativos	36
4.6.1	Frecuencia de despliegue y lead time	36
4.6.2	Tasa de fallos por cambio y tiempo de recuperación	37
4.6.3	Esfuerzo manual del despliegue	38
4.7	Resultados de trazabilidad	38
4.8	Resultados cualitativos	39
4.9	Discusión y síntesis multi-caso	39
CAPÍTULO 5: CONCLUSIONES		41
5.1	Conclusiones respecto del diagnóstico y la brecha inicial	41
5.2	Conclusiones respecto del proceso propuesto	41
5.3	Conclusiones respecto de la validación y comparación con la brecha inicial	42
5.4	Cumplimiento de objetivos	43
5.5	Aportes y contribuciones	44
5.6	Limitaciones	44
5.7	Líneas de trabajo futuro	45
ANEXOS		46
REFERENCIAS BIBLIOGRÁFICAS		48

ÍNDICE DE FIGURAS

1	Árbol de problemas identificado	13
2	Relación SHA (8e7c251) entre ejecución Pipeline y tag imagen docker	48
3	Resultado de ejecución del pipeline CD sobre el <i>commit</i> más reciente en el proyecto B	49

ÍNDICE DE TABLAS

1	Resumen del proceso propuesto: fases, objetivo y evidencia mínima.	24
2	Trazabilidad entre brechas del diagnóstico, fases del proceso y evidencia asociada.	25
3	Fase 1: prácticas mínimas, artefactos y criterios de aceptación.	26
4	Fase 2: prácticas mínimas, artefactos y criterios de aceptación.	27
5	Fase 3: prácticas mínimas, artefactos y criterios de aceptación.	27
6	Fase 4: prácticas mínimas, artefactos y criterios de aceptación.	28
7	Fase 5: prácticas mínimas, artefactos y criterios de aceptación.	29
8	Fase 6: prácticas mínimas, artefactos y criterios de aceptación.	29
9	Caracterización de los proyectos utilizados en la validación (anonimizados).	33
10	Alcance observado por fase del proceso en cada proyecto.	34
11	Métricas utilizadas, definición operativa y fuente primaria.	35
12	Resumen de FD y LT (muestra n=15 para LT).	36
13	Incidentes y MTTR observado — Proyecto A (n=3).	37
14	Incidentes y MTTR observado — Proyecto B (n=4).	37
15	Resumen de CFR y MTTR observado.	38
16	Evidencia cualitativa del esfuerzo manual de despliegue (antes/después).	38
17	Rúbrica de trazabilidad (TRZ) basada en evidencia verificable.	39
18	Muestra de ejecuciones de CD para Proyecto A (n=15).	46
19	Muestra de ejecuciones de CD para Proyecto B (n=15).	47

INTRODUCCIÓN

El presente trabajo aborda el problema de la *ineficiencia y variabilidad* en la entrega y el despliegue de software en equipos pequeños, particularmente en contextos donde los roles tienden a concentrarse en una o dos personas y existe una adopción incompleta de prácticas que, en la literatura, se asocian a ingeniería de software continua y DevOps. En estos entornos, la ausencia de automatización, la baja trazabilidad entre cambios y despliegues, y la limitada sistematización de procesos (por ejemplo, documentación mínima o gestión del trabajo) tienden a afectar tanto la calidad del producto como la capacidad del equipo para iterar con rapidez y control.

Para enfrentar el problema, se realiza un diagnóstico mediante análisis de brecha (*gap analysis*) sobre un conjunto acotado de proyectos descritos en el capítulo de Definición del Problema, utilizando evidencia disponible (repositorios, *pipelines* y registros) y una rúbrica operacional (0–4) para áreas clave asociadas a automatización y trazabilidad. Sobre esa base, se diseña y propone un proceso incremental (una metodología basada en buenas prácticas) estructurado en fases, con artefactos y criterios mínimos verificables. Finalmente, el proceso se valida mediante su aplicación y análisis en dos proyectos reales (Proyecto A y Proyecto B) presentados en el capítulo de Validación de la Solución.

CAPÍTULO 1

DEFINICIÓN DEL PROBLEMA

1.1. Contexto

En la actualidad existe un gran número de empresas de desarrollo de software (DSW) las cuales componen un mercado que se encuentra en auge, haciendo que día a día tengamos nuevas ideas innovadoras propuestas por empresas emergentes, estas empresas suelen tener equipos pequeños de desarrollo, generalmente no especializado, ya que, es necesario que todos sean capaces de aportar en múltiples tareas en los distintos proyectos.

Debido a que en equipos pequeños se tienen roles poco especializados, es frecuente que no se implementen buenas prácticas en el DSW, las cuales pueden incluir: utilizar una estrategia de ramificación¹ de los repositorios *Git*, automatización de procesos de prueba y automatización de procesos de despliegue. La falta de adopción de estas buenas prácticas hace que el proceso de DSW tome mas tiempo del que debería, trayendo problemas en las fechas de entrega debido a problemas que se presentan durante el desarrollo y durante la entrega del producto, por ejemplo, si todas las configuraciones que deben ser realizadas en un servidor al momento de levantar una aplicación deben ser realizadas manualmente en cada entrega al cliente, traería demoras significativas, además, realizar mal una configuración puede generar que toda la aplicación funcione de forma errónea, lo cual deteriora la calidad del software.

Muchas de estas empresas suelen tener presupuestos acotados, debido a esto, las infraestructuras donde sirven² sus aplicaciones para sus clientes suelen encontrarse en servicios en la nube ofrecidos por distintas plataformas, por ejemplo, *Azure*, *Amazon Web Services (AWS)* o *Google Cloud Platform (GCP)*. Se suele optar por contratar servicios en nube, ya que, suelen ser mas económicos, al tener modelos de pago que dependen del uso de los recursos [Veragua Albornoz, 2023], los cuales suelen ser súper especializados en una tarea, siendo estos costos mucho menores a los que puede traer instalar y mantener una infraestructura física dónde se deben tener personal especializado para realizar mantención y disponer de la seguridad y escalabilidad que ofrecen las plataformas mencionadas previamente. Para configurar los servicios que ofrecen estas plataformas en la nube se suele requerir mucho tiempo, porque la configuración se realiza a través de las interfaces que disponen estas plataformas las cuales suelen ser poco intuitivas, se necesita recurrir constantemente a la documentación de cada uno de los recursos utilizados y los tiempos de actualización de estos recursos no suele ser corto, por lo tanto, tener que configurar cada uno de estos servicios cada vez que se debe actualizar la aplicación hace que constantemente se pierda tiempo en una tarea

¹Las estrategias de ramificación son patrones utilizados para gestionar el flujo de trabajo en un repositorio *Git*, esto permite organizar como se crean, modifican y fusionan las ramas.

²Por servir una aplicación nos referimos a disponer la aplicación para uso en producción o en pruebas.

que podría ser automatizada.

Considerando que las empresas suelen alojar sus repositorios de código en plataformas como *GitHub*, *GitLab*, *BitBucket*, *CodeCommit* (Amazon) o *Azure DevOps* (Azure), en las cuales pueden pagar por disponer distintas características como privacidad en los repositorios o tiempo de ejecución de *pipelines*, y no es común que se tengan estos procesos de despliegue o pruebas automatizados se convierte en una pérdida de dinero en suscripciones.

Los equipos pueden adoptar buenas prácticas y así mejorar los tiempos de entrega y la calidad del software, estrategias de ramificación y pipelines³ de testeo o de despliegue permiten un mejor flujo de trabajo durante el desarrollo.

1.2. Diagnóstico y análisis de brecha (gap analysis)

Con el fin de comprender de manera más clara la situación actual en torno a la entrega de software se revisa un conjunto acotado de proyectos/equipos, se plantea un diagnóstico que permita identificar las principales brechas entre las prácticas utilizadas en dichos proyectos y aquellas consideradas como buenas prácticas según el estado del arte. Este diagnóstico se compone de las siguientes etapas:

1.2.1. Alcance del diagnóstico

El análisis se enfoca en una muestra acotada de 4 proyectos/equipos de los cuales se formó parte directamente, con equipos por proyecto de 1 a 6 personas y roles poco especializados por la necesidad de abarcar múltiples tareas. El diagnóstico considera aspectos relacionados con el uso de repositorios *Git*, adopción de estrategias de ramificación, implementación de pipelines de integración continua (CI) y despliegue continuo (CD), además de prácticas vinculadas a la configuración y gestión de infraestructura.

1.2.2. Recolección de información real

Para caracterizar la situación actual, se levantó información de los proyectos considerando los siguientes aspectos, los cuales serán evaluados mediante una escala de 0 a 4 puntos según la implementación de buenas prácticas en cada área:

1. Automatización de despliegue (CD) — 0-4

³Un *pipeline* es una secuencia automatizada de pasos, en la cual podemos asignar tareas que realicen pruebas y despliegues de código.

- Existe un pipeline que despliega automáticamente a algún ambiente (staging/prod) tras merge a la rama principal. (Sí: +1)
- El paso a producción exige gates (aprobación manual o policy) y se permite solo con checks verdes. (Sí: +1)
- El pipeline incluye verificación post-deploy (smoke/health) y frena/revierte si falla. (Sí: +1)
- Se promociona la misma build entre ambientes (artifact promotion) con configuración por entorno (variables/secretos separados). (Sí: +1)

2. Automatización de pruebas (CI) — 0-4

- Hay lint/format automáticos en cada push/PR. (Sí: +1)
- Corren tests automatizados (unit/integration/smoke) en cada PR. (Sí: +1)
- Se ejecuta análisis estático de seguridad/calidad (p. ej., SAST, Sonar). (Sí: +1)
- Los checks son bloqueantes para fusionar/entregar. (Sí: +1)

3. Provisionamiento con IaC — 0-4

- La infraestructura está definida como código y versionada (Terraform/Pulumi/CloudFormation, etc.). (Sí: +1)
- Se realiza plan y revisión (PR/MR) antes de aplicar cambios. (Sí: +1)
- Se pueden recrear entornos (al menos dev y staging) de forma reproducible con IaC. (Sí: +1)
- La provisión IaC está integrada al pipeline (validate/plan/apply con gates). (Sí: +1)

4. Trazabilidad y registro del proceso — 0-4

- Cada despliegue registra commit SHA/versión de artefacto. (Sí: +1)
- Los logs del pipeline quedan persistentes con marcas de tiempo de inicio/fin. (Sí: +1)
- Existe un mecanismo visible de versión por ambiente (endpoint/tag/label). (Sí: +1)
- Hay correlación PR, build y deploy (IDs enlazados/trazabilidad automática). (Sí: +1)

5. Propiedad del despliegue — 0-4

- El despliegue lo ejecuta y mantiene el equipo interno (no un tercero). (Sí: +1)
- Hay roles responsables definidos (release manager/on-call) para desplegar y resolver incidentes. (Sí: +1)
- Existe documentación (runbook) de despliegue accesible y vigente. (Sí: +1)

- Al menos dos personas pueden ejecutar el proceso (sin punto único de falla). (Sí: +1)

6. Lead time de cambio (umbral acumulativo) — 0-4

- Lead time $\leq$ 7 días. (Sí: +1)
- Lead time $\leq$ 3 días. (Sí: +1)
- Lead time $\leq$ 8 horas. (Sí: +1)
- Lead time $\leq$ 1 hora. (Sí: +1)

Usar un periodo representativo (p. ej., últimos 3 cambios comparables). Si no se dispone de trazas, marcar “no observable”.

7. Tasa de fallos de cambio (CFR) — 0-4

- CFR $\leq$ 30 %. (Sí: +1)
- CFR $\leq$ 20 %. (Sí: +1)
- CFR $\leq$ 10 %. (Sí: +1)
- CFR $\leq$ 5 %. (Sí: +1)

Si no hay datos, utilizar estimación cualitativa con nota de confianza.

8. MTTR / rollback — 0-4

- Existe procedimiento documentado de rollback/recovery. (Sí: +1)
- El rollback es ejecutable mediante script o comando único (no ad-hoc). (Sí: +1)
- MTTR típico $\leq$ 4 horas para incidentes de despliegue. (Sí: +1)
- MTTR típico $\leq$ 30 minutos (o rollback casi inmediato). (Sí: +1)

Se observa lo siguiente en los proyectos analizados:

1. Proyecto A - “IaC + CD para clientes (Azure)”

- Identificación y alcance: sin fechas formales, reuniones semanales desde el mes 1. Solución crítica (IaC + CD para servir la app). Múltiples entornos previstos, solo dev activo usado “como prod”.
- Equipo y roles: un integrante. sin responsable de releases. Solo una persona con capacidad de despliegue. Antes del pipeline, los despliegues eran manuales por el desarrollador de la aplicación. La IaC inicialmente era ejecutada por el equipo, hacia el final, mediante Terraform Cloud.
- Arquitectura y stack: Terraform en Azure. AKS como destino. Azure DevOps para pipelines. Terraform Cloud como pipeline de IaC.

- Flujo de trabajo: repos en Azure DevOps. Sin gestor de issues. Ramas *dev/prod*, commits mayormente en *dev*. *prod* no usado.
- CI: no se encontraba implementada.
- CD: IaC con despliegue automático en fases finales (trigger por commit en *dev*). Aprobación funcional del cliente antes de ejecutar Terraform. *Terraform Cloud* mostraba errores de *apply*.
- IaC: no existía previamente. Código pensado para multiambiente, pero solo *dev* activo. Integración al pipeline al final.
- Trazabilidad (TRZ): sin artefactos/versionado visible por ambiente. Logs de pipeline con duración. Sin correlación entre PR/commit, *build* y despliegue.
- Métricas operativas: lead time típico ≥ 3 días (aprobaciones/gestiones). Frecuencia ~ 1 /semana. CFR No observado (N/O). MTTR N/O.
- Seguridad y control: secretos gestionados en Azure DevOps y/o Key Vault. Acceso mediante procesos específicos (API keys, permisos).
- Evidencia disponible hoy: evidencia testimonial del equipo. Hito relevante: incorporación de Terraform Cloud. Nivel de confianza alto.
- Restricciones y contexto: uso de cuentas del cliente. Tramitación de permisos, direcciones IP y firewall para habilitar servicios.

2. Proyecto B - "Módulo frontend en SaaS"

- Identificación y alcance: involucrado ~ 1 mes. Producto > 1 año en prod. Módulo nuevo aún no usado en prod. Entorno local. En repos solo un entorno.
- Equipo y roles: tres integrantes (PM, desarrollador principal y colaborador). Despliegue a cargo de un tercero externo. Demoras de 1 día a 1 semana para calendarizar. Sin responsable de releases. Solo el externo realizaba el despliegue.
- Arquitectura y stack: microservicio de un producto mayor. Hosting desconocido. Base y BD críticas.
- Flujo de trabajo: uso probable de Jira, sin utilización consistente para las tareas. Estrategia de ramas no establecida (commits en una única rama).
- CI: no se encontraba implementada.
- CD: existía despliegue automático "opaco" (externo, sin documentación). El equipo no conocía ni controlaba el proceso.
- IaC: N/O.
- Trazabilidad (TRZ): sin SHAs/artefactos visibles. Sin mecanismo de versión por ambiente. Logs de despliegue no accesibles.
- Métricas operativas: lead time típico > 1 día, usualmente ≥ 3 días. Frecuencia ~ 1 /semana (podía variar). CFR N/O. MTTR N/O.
- Seguridad y control: información no disponible.

- Evidencia disponible hoy: principalmente evidencia testimonial. Alto nivel de confianza. Afirmaciones marcadas como hipotéticas corresponden a confianza media.
- Restricciones y contexto: sin acceso al proceso de despliegue.

3. Proyecto B - “microservicios, Azure + servidor propio”

- Identificación y alcance: ~ 1 año. Web tipo SaaS. Entornos *dev* y *prod* activos.
- Equipo y roles: 6 (marketing, web, backend, IA, infra/app, PO). Despliegue interno. Sin responsable de releases y solo 2 personas sabían desplegar.
- Arquitectura y stack: microservicios con despliegue independiente y versionado propio. Hosting mixto (Azure VM como host de contenedores + servidor de un participante).
- Flujo de trabajo: Jira con issues y responsables. Estrategia Gitflow.
- CI: tests básicos en backend. Sin pipeline de CI.
- CD: pipeline CD por microservicio. Deploy tras merge a ramas *dev/main*. Aprobación manual por un integrante. Verificación manual post-deploy. Sin promoción de mismo artefacto entre entornos.
- IaC: no había.
- Trazabilidad (TRZ): deploy sin registro explícito de SHA/artefacto. asociado al commit pero no al nombre de imagen. Logs de deploy con duraciones. Sin versión visible por ambiente.
- Métricas operativas: lead time < 1 día (a veces < 1 h si los revisores están disponibles). frecuencia según demanda del sprint. CFR no registrado. MTTR $\sim < 1$ h con rollback (mayor si requiere corrección).
- Seguridad y control: secretos gestionados en servicios de secretos/pipelines. Acceso a infra limitado a quienes desplegaban.
- Evidencia disponible hoy: Jira, repos, chats, logs de pipelines. Confianza alta.
- Restricciones/contexto: plazos académicos. Algunas funcionalidades condicionadas por requisitos de la universidad.

4. Proyecto C - “Método de desarrollo en empresa (equipos de 1 dev)”

- Identificación y alcance: varios proyectos, la mayoría > 1 año y vigentes. SaaS o módulos de SaaS. típicamente solo *prod* y local del dev.
- Equipo y roles: generalmente 1 dev + responsable general. El dev también trata con cliente/PO. Cada dev despliega su proyecto.
- Arquitectura y stack: microservicios. Nube principalmente AWS. Dependencias críticas (IoT, BD) según proyecto.
- Flujo de trabajo: de Jira a Linear. Muchos proyectos sin estrategia Git definida. Algunos con prácticas tipo Gitflow.

- CI: inexistente en la mayoría (sin testing automatizado).
- CD: algunos proyectos con pipelines CD (trigger por commit/PR/merge a *main*). En general sin aprobaciones (un solo dev). Sin promoción de artefacto entre entornos (no hay multiambiente).
- IaC: presente en pocos proyectos nuevos. Versionada, *plan/apply* ejecutado por el dev, entornos reproducibles no establecidos aunque el código admite variables por ambiente. IaC con su propio pipeline en los casos que existe.
- Trazabilidad (TRZ): solo un proyecto registra commit SHA/artefacto. los que tienen CD guardan logs de despliegue y duración. Sin versión visible por ambiente.
- Métricas operativas: lead time usual < 1 día (dominado por tiempo de desarrollo). Frecuencia variable (desde varios por semana hasta por sprint). CFR no definido. MTTR no medido.
- Seguridad y control: gestión de secretos dispar (algunos en archivos, otros en servicios de secretos). Accesos a infra limitados a recursos necesarios.
- Evidencia disponible hoy: correos, chats, repos, logs de pipelines, Linear/Jira. Confianza alta.
- Restricciones/contexto: plazos de clientes/responsables.
- Evidencia disponible hoy: correos, chats, repos, logs de pipelines, Linear/Jira. Confianza alta.
- Restricciones/contexto: plazos de clientes/responsables.

1.2.3. Análisis de brecha

El análisis de brecha permite identificar carencias entre las prácticas actuales observadas y las metodologías consolidadas en el DSW. Este contraste es importante porque revela puntos críticos que afectan eficiencia, calidad y trazabilidad, y sirve como base para una adopción incremental de buenas prácticas. En cada proyecto se contrasta el estado actual con el estado del arte en gestión de código, pruebas, despliegue e infraestructura, para evidenciar dónde cerrar brechas y con qué prioridad.

Metodología y criterios

- **Alcance del análisis:** se evalúan áreas clave: Git/ramificación, CI, CD, IaC/CaC, trazabilidad, seguridad/accesos y métricas operativas (LT, CFR, MTTR).
- **Fuentes y evidencia:** repositorios, pipelines, logs de despliegue y documentación disponible, según el nivel de acceso en cada caso.

- **Buenas prácticas de referencia:** estrategias de ramas (Gitflow, GitHub Flow, Trunk-Based) [Ram, 2013], pipelines de CI con pruebas y quality gates [Cowell et al., 2023], CD con verificaciones y reversión [Zampetti et al., 2021], e Infraestructura/Configuración como código (Terraform, Ansible, etc.) [Morris, 2020].
- **Priorización de cierre:** *Alta/Media/Baja* según impacto en tiempo de entrega y calidad, riesgo operativo, esfuerzo estimado y dependencias externas.
- **Lectura de matrices:** cada fila muestra la situación actual, la práctica recomendada y la brecha/prioridad de cierre. Las matrices se organizan por proyecto para facilitar trazabilidad.

A continuación se incluyen las matrices de brecha por proyecto, manteniendo la narrativa para contextualizar el análisis y guiar la priorización de acciones.

Proyecto A — IaC + CD para clientes (Azure)

Git/ramificación Situación actual: repos en Azure DevOps. Ramas *dev/prod*. Commits principalmente en *dev*. *prod* no usado. Sin gestor de issues ni PR obligatorias. Recomendado: convención de ramas (Gitflow/Trunk), PR obligatorias con revisores, etiquetado semántico y protección de *main/prod*. Baseline: N/O. Brecha: N/O.

CI Situación actual: inexistente. Recomendado: pipeline CI con lint/format, pruebas unitarias/integración/e2e, SAST y checks bloqueantes en PR. Baseline: 0/4. Brecha: 4.

CD Situación actual: IaC con despliegue automático a *dev*. Aprobación funcional previa. Sin promoción del mismo artefacto ni gates a producción. Recomendado: CD a *staging* tras merge a *main* con gates, promoción del mismo artefacto entre ambientes, verificación smoke/health y rollback automatizado. Baseline: 1/4. Brecha: 2.

IaC/CaC Situación actual: Terraform versionado y en Terraform Cloud. Diseño multiamiente, pero solo *dev* activo. Recomendado: parametrizar entornos reproducibles (*dev/staging/prod*), integrar *validate/plan/apply* con controles y revisión por PR. Baseline: 3/4. Brecha: 0.

Trazabilidad (TRZ) Situación actual: sin registro de SHA/artefacto por despliegue. Sin versión visible por ambiente. Logs de pipeline con duración. Sin correlación entre PR, build y despliegue. Recomendado: registrar SHA y versión por despliegue, mecanismo visible por ambiente (endpoint/tag/label) y enlazar PR, build y despliegue. Baseline: 1/4. Brecha: 2.

Seguridad y acceso Situación actual: secretos en Azure DevOps y/o Key Vault. Accesos vía procesos específicos. Recomendado: centralizar gestión de secretos con rotación y *least privilege*. Auditoría de accesos. Separar permisos de deploy/infra. Baseline: N/O. Brecha: N/O.

Métricas (LT, CFR, MTTR) Situación actual: LT típico ≥ 3 días. Frecuencia ~ 1 /semana. CFR y MTTR N/O. Recomendado: instrumentar métricas DORA (LT ≤ 8 h, CFR ≤ 10 %, MTTR ≤ 4 h) y pruebas de rollback. Baseline: 1/4 (LT). Brecha: 2 (LT).

Proyecto B — Módulo frontend en SaaS

Git/ramificación Situación actual: estrategia no establecida. Commits en una sola rama. Uso inconsistente de gestor (posiblemente Jira). Recomendado: definir estrategia (Git-flow/GitHub Flow), PR obligatorias con revisores, protecciones de rama y etiquetado de versiones. Baseline: 0/4. Brecha: 3.

CI Situación actual: inexistente. Recomendado: pipeline CI con lint/format, pruebas unitarias/integración/e2e y checks bloqueantes. Baseline: 0/4. Brecha: 4.

CD Situación actual: deploy “automático” externo y opaco. Equipo sin control ni documentación del proceso. Un solo entorno en repos. Recomendado: internalizar CD con visibilidad y control. Ambientes staging/prod. Aprobaciones/gates. Promoción del mismo artefacto. Baseline: 0/4. Brecha: 3.

IaC/CaC Situación actual: N/O. Recomendado: adoptar IaC/CaC (Terraform/Ansible) versionado e integrado al pipeline. Baseline: 0/4. Brecha: 3.

Trazabilidad (TRZ) Situación actual: sin SHAs/artefactos por deploy. Sin logs accesibles. Sin mecanismo de versión por ambiente. Recomendado: registrar SHA/artefacto por deploy, logs persistentes y versión visible por ambiente. Baseline: 0/4. Brecha: 3.

Seguridad y acceso Situación actual: sin información. Recomendado: política de secretos (servicio dedicado) y control de accesos con auditoría. Baseline: N/O. Brecha: N/O.

Métricas (LT, CFR, MTTR) Situación actual: LT > 1 día (usualmente ≥ 3 días). Frecuencia ~ 1 /semana. CFR/MTTR N/O. Recomendado: instrumentar DORA (LT ≤ 8 h, CFR ≤ 10 %, MTTR ≤ 4 h) y establecer rollback. Baseline: 1/4 (LT). Brecha: 2 (LT).

Proyecto C — microservicios, Azure + servidor propio

Git/ramificación Situación actual: estrategia Gitflow en uso. Jira con issues y responsables. Recomendado: mantener estrategia con PR obligatorias y protecciones. Evaluar simplificación Trunk si reduce ciclo sin perder control. Baseline: N/O. Brecha: N/O.

CI Situación actual: tests básicos en backend, sin pipeline CI ni checks bloqueantes. Recomendado: pipeline CI completo con análisis estático y gates. Baseline: 0/4. Brecha: 4.

CD Situación actual: pipelines por microservicio. Deploy tras merge a *dev/main*. Aprobación manual. Verificación manual. Sin promoción del mismo artefacto. Recomendado: promoción del mismo artefacto entre entornos, verificaciones automáticas smoke/health y rollback automatizado. Baseline: 3/4. Brecha: 0.

IaC/CaC Situación actual: no se utilizó IaC. Recomendado: adoptar IaC/CaC para reproducibilidad y control de cambios (Terraform/Ansible) integrado a pipeline. Baseline: 0/4. Brecha: 3.

Trazabilidad (TRZ) Situación actual: deploy sin SHA/artefacto registrado. Sin versión visible por ambiente. Sin correlación entre PR, build y despliegue. Existen logs de duración. Recomendado: registro de SHA/artefacto, versión visible por ambiente y correlación entre PR, build y despliegue. Baseline: 1/4. Brecha: 2.

Seguridad y acceso Situación actual: secretos gestionados en servicios/pipelines. Acceso a infra limitado a quienes despliegan. Recomendado: reforzar *least privilege*, rotación de secretos y auditoría. Formalizar roles de deploy y acceso. Baseline: N/O. Brecha: N/O.

Métricas (LT, CFR, MTTR) Situación actual: LT predominante < 1 día. CFR no registrado. MTTR $\sim < 1$ h con rollback manual. Recomendado: medir CFR. Formalizar rollback automatizado. Objetivos $LT \leq 8$ h, $CFR \leq 10\%$, $MTTR \leq 4$ h. Baseline: 2/4 (LT) y 1/4 (MTTR). Brecha: 1 (LT) y 2 (MTTR).

Proyecto D — Método de desarrollo en empresa (equipos de 1 dev)

Git/ramificación Situación actual: muchos proyectos sin estrategia definida. Algunos con prácticas tipo Gitflow. Gestión de trabajo de Jira a Linear. Recomendado: convenciones de ramas/commits, PR obligatorias y protecciones. Etiquetado de versiones. Baseline: N/O. Brecha: N/O.

CI Situación actual: inexistente en la mayoría (sin testing automatizado). Recomendado: establecer CI uniforme con lint/format, unit/integration/e2e y gates. Baseline: 0/4. Brecha: 4.

CD Situación actual: algunos proyectos con CD (trigger por commit/PR/merge a *main*). Generalmente sin aprobaciones ni multiambiente. Recomendado: CD estándar con ambientes reproducibles y gates/aprobaciones. Promoción del mismo artefacto. Baseline: 1/4. Brecha: 2.

IaC/CaC Situación actual: presente en pocos proyectos nuevos. Versionada, *Validate/plan/apply* ejecutado por el *dev*. Entornos reproducibles no establecidos. Recomendado: ampliar IaC/CaC para proyectos críticos. Parametrización por ambiente y pipeline propio. Baseline: 0/4. Brecha: 3.

Trazabilidad (TRZ) Situación actual: solo un proyecto registra commit SHA/artefacto. Los que tienen CD guardan logs. Sin versión visible por ambiente. Recomendado: registro de SHA/artefacto y versión visible. Conservar logs de pipeline con correlación. Baseline: 1/4. Brecha: 2.

Seguridad y acceso Situación actual: gestión de secretos dispar (archivos vs servicios de secretos). Accesos a infra limitados a recursos necesarios. Recomendado: centralizar secretos en servicio dedicado. Rotación y *least privilege*. Auditoría de accesos. Baseline: N/O. Brecha: N/O.

Métricas (LT, CFR, MTTR) Situación actual: LT usual < 1 día. CFR no definido. MTTR no medido. Recomendado: medir DORA (LT, CFR, MTTR) y objetivos $LT \leq 8$ h, $CFR \leq 10\%$, $MTTR \leq 4$ h. Baseline: 2/4 (LT). Brecha: 1 (LT).

1.2.4. Priorización de problema

Se busca abordar principalmente la ausencia de automatización en los procesos de desarrollo y entrega de software dentro de empresas pequeñas, considerando como muestra de estudio los 4 casos mencionados anteriormente, así mismo aumentar la eficiencia en los tiempos de entrega mediante la implementación de las prácticas y tecnologías que permitan la automatización de estos procesos, haciendo que sean flujos sistematizados y reproducibles. De esta forma, se pretende optimizar los recursos disponibles, mejorar la calidad del software entregado y fortalecer la competitividad de las empresas en el mercado.

Como cierre del diagnóstico, este trabajo culmina en el diseño de un proceso basado en buenas prácticas para automatizar la entrega y el despliegue de software, que sirva como receta aplicable de forma incremental en equipos o proyectos similares a los analizados.

1.3. Causas y efectos del problema

Las distintas causas y efectos del problema se pueden observar en el árbol del problema realizado (Ver figura 1).

1.3.1. Problema central

Como problema central tenemos la "entrega de software ineficiente en empresas emergentes de DSW". Este problema genera deficiencias en el producto de software en el cual la empresa está trabajando.

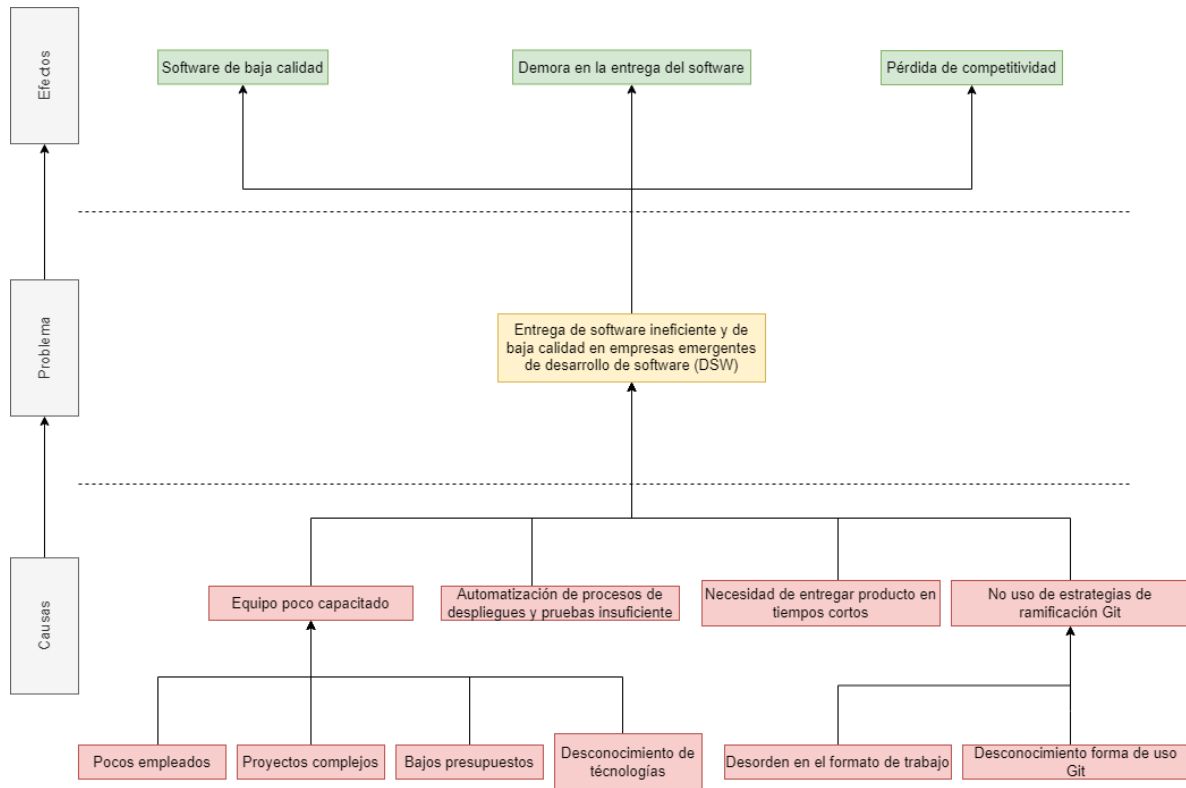


Figura 1: Árbol de problemas identificado

Fuente: Elaboración propia

1.3.2. Causas

Como las causas del problema encontramos:

- **Equipo poco capacitado:** Los equipos de desarrollo son pequeños y usualmente desarrollan proyectos complejos en los cuales cada uno de los integrantes debe desempeñar más de un rol teniendo así un equipo poco especializado que no tiene conocimiento de las mejores prácticas o tecnologías actuales.
- **Automatización de procesos insuficientes:** Muchos procesos se realizan de forma manual debido al desconocimiento o poca planificación en el desarrollo.
- **Tiempos de entrega acotados:** Debido a los cortos plazos de entrega, se recurre frecuentemente a metodologías de desarrollo ágiles como una alternativa para acelerar los ciclos de trabajo [Malla, 2024]. Sin embargo, equipos reducidos y poco especializados privilegian la rapidez sobre la implementación de buenas prácticas.
- **Uso poco refinado de Git:** Se utiliza Git sin conocer o aplicar métodos probados o recomendados.

1.3.3. Efectos

Los efectos del problema son:

- Software de baja calidad: Se genera un software que puede considerarse deficiente, con errores o que no cumpla correctamente los requisitos solicitados.
- Demora en la entrega: Los plazos pueden no cumplirse o cumplirse teniendo a los desarrolladores más tiempo del estimado trabajando en las distintas características solicitadas.
- Pérdida de competitividad: Se obtiene una baja capacidad en responder a las necesidades de los clientes a tiempo, de forma frecuente o de mayor calidad, lo cual limita las oportunidades de crecimiento en el mercado de desarrollo.

1.4. Prácticas adoptables

Las prácticas adoptables para cerrar las brechas detectadas se sintetizan a continuación, en clave de problema-acción-resultado esperado:

- Estrategias de ramificación (Git): Definir convención de ramas/commits, uso de solicitudes de cambio y etiquetado de versiones para habilitar trabajo paralelo sin bloqueos y mayor trazabilidad. Resultado esperado: menor fricción al integrar y mejor control del flujo de cambios. Indicadores: conflictos por fusión, tiempo medio de integración, lead time de cambios.
- Integración continua (CI): Ejecutar automáticamente compilación, pruebas y análisis en cada cambio antes de fusionar a la rama principal. Resultado esperado: detección temprana de defectos y reducción de regresiones. Indicadores: tasa de fallos post-merge, tiempo de detección/corrección, porcentaje de builds verdes.
- Entrega/Despliegue continuo (CD): Automatizar empaquetado y despliegue a preproducción y producción con mecanismos de reversión. Resultado esperado: liberaciones más frecuentes y de menor riesgo. Indicadores: frecuencia de despliegue, tiempo para restaurar servicio, tasa de despliegue fallido.
- Infraestructura y configuración como código (IaC/CaC): Versionar y automatizar aprovisionamiento y configuración (por ejemplo, Terraform/Ansible) integrados al pipeline. Resultado esperado: entornos reproducibles, menos errores manuales y menor tiempo de preparación. Indicadores: tiempo de aprovisionamiento, incidentes por cambios manuales, deriva de configuración.

Estas prácticas se incorporarán de manera incremental en un proyecto piloto, priorizando quick wins, manteniendo criterios de aceptación claros por cada incorporación y finalmente realizando una comparación con las brechas detectadas previamente.

1.5. Alcances y limitaciones

Alcances: El diagnóstico se realiza sobre una muestra acotada de 4 proyectos/equipos en los cuales hubo participación, utilizando evidencia disponible (repositorios, pipelines, logs y testimonios) y contrastando con buenas prácticas del estado del arte.

Limitaciones: Este estudio no pretende representar ni resolver la totalidad de la industria. Los resultados y recomendaciones se circunscriben al conjunto analizado y a contextos con características comparables. Algunas métricas se estiman con la mejor evidencia disponible cuando no existen trazas completas.

1.6. Objetivos

A partir de lo mencionado anteriormente se presentan los siguientes objetivos:

1.6.1. Objetivo general

Diseñar un proceso basado en buenas prácticas para automatizar el despliegue y la entrega de software en equipos pequeños, utilizando como insumo el diagnóstico y las brechas identificadas en la muestra analizada.

1.6.2. Objetivos específicos

- Realizar un análisis de brecha sobre distintas prácticas de desarrollo y entrega de software en un conjunto acotado de proyectos/equipos.
- Diseñar el proceso propuesto, estructurando fases, prácticas mínimas y criterios de aceptación para adopción incremental.
- Aplicar el proceso en un proyecto piloto seleccionado del conjunto analizado.
- Evaluar resultados del piloto comparando antes vs. después (lead time, frecuencia de despliegue, CFR, MTTR, trazabilidad) y documentar lecciones para su ajuste y generalización.

CAPÍTULO 2

MARCO CONCEPTUAL

2.1. Metodologías Ágiles

Las metodologías ágiles son un conjunto de prácticas y valores para el desarrollo de software que priorizan la flexibilidad, la iteración rápida y la respuesta a cambios. Según [Fowler et al., 2001], el Manifiesto Ágil establece cuatro valores fundamentales: valorar a los individuos e interacciones sobre procesos y herramientas, software funcionando sobre documentación completa, colaboración con el cliente sobre negociación de contratos, y responder a cambios sobre seguir un plan.

[Dingsøyr et al., 2012] señala que las metodologías ágiles han evolucionado significativamente durante la última década, transformando la forma en que los equipos de desarrollo abordan la creación de software. Estas metodologías permiten que los equipos se adapten rápidamente a los requisitos cambiantes del cliente y a las nuevas condiciones del mercado.

Sin embargo, [Nerur et al., 2005] destacan que la migración a metodologías ágiles presenta desafíos significativos para las organizaciones, incluyendo cambios culturales profundos, necesidad de capacitación del personal y la reorganización de procesos establecidos. A pesar de estos desafíos, la adopción de metodologías ágiles ha demostrado mejorar la calidad del software, reducir los tiempos de entrega y aumentar la satisfacción del cliente. En el contexto de esta memoria, su adopción ofrece un marco para mejorar la eficiencia y previsibilidad de la entrega en equipos pequeños y con múltiples responsabilidades.

2.1.1. Desarrollo dirigido por tests (TDD)

El Desarrollo Dirigido por Tests (Test-Driven Development, TDD) es una técnica de diseño y construcción que invierte el orden tradicional: primero se especifica el comportamiento deseado como una prueba que falla, luego se implementa el mínimo código para pasarla y finalmente se refactoriza preservando el comportamiento [Beck, 2002]. Este ciclo Red-Green-Refactor guía decisiones de diseño y favorece la simplicidad.

Además del ciclo básico, TDD se apoya en prácticas de diseño guiadas por pruebas para hacer crecer software orientado a objetos mediante pequeñas iteraciones, con énfasis en colaboración entre objetos, uso de dobles de prueba y diseño emergente [Freeman and Pryce, 2009]. La calidad de las pruebas y su mantenibilidad se sustentan en patrones y “smells” del código de test (por ejemplo, *Obscure Test*, *Fragile Test*, *Test Code Duplication*) y técnicas de refactorización del código de prueba [Meszaros, 2007]; guías prácticas complementan la aplicación en contextos reales [Astels, 2003].

La evidencia empírica sobre TDD muestra resultados positivos en calidad externa y efectos variables en productividad según el contexto organizacional y la experiencia del equipo: estudios controlados y metaanálisis reportan mejoras en defectos y mantenibilidad, con impacto en velocidad dependiente de la curva de aprendizaje y la disciplina del equipo [Erdogmus et al., 2005, Rafique and Mišić, 2012, Janzen and Saiedian, 2005]. En entornos ágiles, TDD actúa como mecanismo de especificación ejecutable y facilita refactorización segura, reforzando la entrega frecuente de valor.

2.2. DevOps

DevOps es un conjunto de prácticas, herramientas y una cultura que busca mejorar la colaboración entre los equipos de desarrollo (Dev) y operaciones (Ops) para acelerar la entrega de software y mejorar su calidad. Según [Ebert et al., 2016], DevOps integra actividades de desarrollo, prueba y operaciones para automatizar y simplificar el ciclo de vida del software, eliminando los silos tradicionales entre equipos.

[Bass et al., 2015] proporciona una perspectiva arquitectónica de DevOps, enfatizando que esta no es solo una cuestión de herramientas o procesos, sino de un cambio cultural y organizacional fundamental. DevOps requiere que los equipos compartan responsabilidades desde el desarrollo hasta la ejecución en producción, fomentando la comunicación constante, la automatización integral y la mejora continua.

Los pilares de DevOps incluyen la automatización de procesos, la medición y monitoreo del sistema, y la retroalimentación rápida. DevOps permite que las organizaciones entreguen cambios de software más frecuentemente, reduzcan el tiempo entre entregas, minimicen los errores introducidos en producción y mejoren la confiabilidad general del sistema. Este enfoque es especialmente valioso para empresas pequeñas con equipos reducidos, ya que permite que realicen tareas que típicamente requerirían múltiples especialistas, optimizando recursos y mejorando la eficiencia operativa.

2.2.1. Integración continua (CI)

La Integración Continua (Continuous Integration, CI) es una práctica de ingeniería de software en la que los cambios de código se integran con frecuencia en un repositorio compartido y, de forma automática, se ejecutan procesos de construcción y pruebas para detectar fallos tempranamente. Según [Meyer, 2014], CI se apoya en servidores y herramientas que automatizan la compilación, la ejecución de suites de pruebas y la notificación de resultados, con el objetivo de acortar el ciclo de retroalimentación y reducir los problemas de integración.

Estudios recientes destacan que la CI suele materializarse en pipelines que orquestan etapas como compilación, pruebas unitarias y de integración, análisis estático, empaque-

tado y publicación de artefactos, activándose ante cada commit o solicitud de cambio [Zampetti et al., 2021]. Esta automatización incrementa la calidad del software, facilita la refactorización segura y habilita cadencias de entrega más rápidas, proporcionando además trazabilidad y visibilidad del estado del proyecto. Revisiones sistemáticas sintetizan enfoques, herramientas y desafíos recurrentes en CI/CD que refuerzan estas prácticas y su adopción gradual en la industria [Shahin et al., 2017].

2.2.2. Entrega y despliegue continuos (CD)

La Entrega y Despliegue Continuos (Continuous Delivery/Deployment, CD) extiende la Integración Continua para asegurar que el software esté siempre en un estado desplegable y que los cambios puedan liberarse de forma frecuente y confiable. Según [Humble and Farley, 2010], CD se apoya en un pipeline que automatiza construcción, pruebas (unitarias, de integración y de aceptación), empaquetado y preparación para la liberación, manteniendo la calidad en cada etapa y reduciendo el riesgo; estudios de revisión amplían el panorama de enfoques, herramientas y prácticas asociadas [Shahin et al., 2017].

Existen dos modalidades complementarias:

- Entrega continua (continuous delivery): el pipeline deja el sistema listo para producción; la publicación a producción es una decisión manual, aunque técnicamente siempre viable [Humble and Farley, 2010].
- Despliegue continuo (continuous deployment): cuando todas las verificaciones del pipeline pasan, el despliegue a producción ocurre automáticamente, habilitando cadencias aún más rápidas y requiriendo altos niveles de automatización, monitoreo y mecanismos de reversión [Leppänen et al., 2015].

Prácticas clave incluyen versionar todo (código, dependencias y configuraciones), mantener paridad entre entornos (desarrollo, preproducción y producción), automatizar despliegues y reversión (rollback), emplear pruebas automatizadas en múltiples niveles y utilizar conmutadores de funcionalidades (feature toggles) para activar/desactivar capacidades sin desplegar nuevo código [Shahin et al., 2017]. La adopción también presenta desafíos técnicos y sociales, con problemas típicos y mitigaciones reportadas en la literatura [Claps et al., 2015, Laukkanen et al., 2017].

Los beneficios de CD incluyen tiempos de entrega más cortos, liberaciones más predecibles y confiables, menor deuda operativa y una reducción del riesgo asociado a cambios grandes, habilitando ciclos de retroalimentación más rápidos con usuarios y negocio.

2.2.3. Infraestructura como código (IaC)

La Infraestructura como Código (Infrastructure as Code, IaC) es el enfoque de gestionar y aprovisionar infraestructura mediante archivos de definición y código declarativo o imperativo, en lugar de procesos manuales. Según [Morris, 2016, Morris, 2020], IaC permite versionar, automatizar y reproducir entornos de manera consistente a través de distintas etapas (desarrollo, pruebas, producción), reduciendo la configuración manual y los errores asociados.

Herramientas como Terraform promueven un estilo declarativo en el cual se especifica el estado deseado de los recursos y el motor resuelve las acciones necesarias para alcanzarlo, habilitando prácticas de validación, módulos reutilizables y flujos de cambio revisables [Brikman, 2022]. En organizaciones modernas, IaC se integra en pipelines CI/CD para validar, planificar y aplicar cambios de infraestructura con controles de calidad y seguridad, acelerando la entrega y manteniendo trazabilidad [Jayaram et al., 2024].

2.2.4. Configuración como código (CaC)

La Configuración como Código (Configuration as Code, CaC) es la práctica de definir en código versionable el estado deseado de sistemas y aplicaciones —paquetes, archivos, servicios, usuarios, permisos y parámetros— y aplicarlo de forma repetible e idempotente. Herramientas de gestión de configuración como Ansible permiten describir playbooks y roles declarativos para configurar servidores y aplicaciones sobre inventarios de máquinas, manteniendo la convergencia hacia el estado objetivo y evitando la deriva de configuración [Hochstein and Moser, 2017].

CaC complementa a IaC: mientras IaC aprovisiona los recursos de infraestructura, CaC establece y mantiene su configuración de sistema y de aplicaciones. Integrado en pipelines CI/CD, CaC habilita revisiones de cambios, pruebas en entornos de preproducción (staging) y despliegues consistentes y auditables, disminuyendo errores manuales y reduciendo el tiempo de recuperación ante fallos.

2.3. Control de versiones con Git

Git es un sistema de control de versiones distribuido que permite registrar la historia de cambios en el código fuente, habilitar trabajo paralelo mediante ramas, fusionar contribuciones y recuperar estados anteriores con precisión. Su modelo distribuido y sus operaciones ligeras facilitan la colaboración entre equipos, mejoran la trazabilidad y soportan prácticas de revisión y auditoría del código [Chacon and Straub, 2014]. En entornos científicos y de ingeniería, el uso disciplinado de Git incrementa la reproducibilidad y la transparencia de los

resultados al vincular versiones de datos, scripts y documentación con commits verificables [Ram, 2013].

Buenas prácticas ampliamente difundidas incluyen realizar commits pequeños y atómicos con mensajes descriptivos, usar ramas para aislar características o correcciones, abrir solicitudes de cambio (pull requests) para revisión por pares y aprovechar plataformas como GitHub para gestionar incidencias (issues) e integrar automatizaciones [Perez-Riverol et al., 2016]. Estas prácticas proporcionan trazabilidad de decisiones, facilitan la resolución de conflictos y habilitan la integración con pipelines CI/CD para validar y desplegar cambios de forma confiable.

2.3.1. Estrategias de ramificación

Una estrategia de ramificación es el conjunto de reglas y convenciones que define cómo un equipo crea, nombra, integra y elimina ramas en un repositorio Git, así como cuándo libera versiones y cómo gestiona correcciones urgentes. Su objetivo es coordinar trabajo en paralelo, reducir el riesgo de integración, asegurar trazabilidad y sostener una cadencia de entrega compatible con CI/CD. Típicamente, una estrategia especifica: tipos de ramas permitidas, criterios de apertura y cierre, políticas de fusión y revisión (por ejemplo, pull requests), relación con ambientes (pruebas/producción) y disparadores de pipelines.

La elección de estrategia depende del tamaño del equipo, la frecuencia de liberaciones, la criticidad del producto y requisitos de cumplimiento. Existen enfoques más ligeros y otros de mayor control; entre los más documentados se encuentran Git Flow, Trunk-Based Development y los flujos GitHub/GitLab Flow, que representan distintas compensaciones entre estabilidad y velocidad [Driessen, 2010, Hammant, GitHub, GitLab, Forsgren et al., 2018].

2.4. Infraestructura Cloud

La infraestructura cloud se basa en la provisión de recursos computacionales (cómputo, almacenamiento y redes) como servicios bajo demanda a través de Internet, con elasticidad, medición del uso y modelos de servicio como IaaS, PaaS y SaaS, así como diferentes modelos de despliegue (público, privado e híbrido) [Buyya et al., 2011]. Este paradigma permite a las organizaciones ajustar capacidad según necesidad y optimizar costos operativos al adoptar un esquema de pago por uso.

La adopción de cloud presenta oportunidades relevantes y, al mismo tiempo, desafíos de seguridad. Entre las principales consideraciones se encuentran la confidencialidad e integridad de los datos, la disponibilidad de los servicios, el aislamiento en entornos multiarrendatario, la gestión de identidades y el cumplimiento normativo [Ali et al., 2015]. En escenarios donde se requiere mayor control y gobierno, las guías para construir nubes privadas recomiendan

estandarizar plataformas, automatizar el aprovisionamiento y definir procesos de operación para asegurar consistencia y eficiencia [Babar and Pantić, 2012].

Desde la perspectiva de cargas intensivas de datos, la infraestructura cloud habilita asignación elástica de recursos y estrategias de planificación para dar soporte a aplicaciones de big data, atendiendo a requisitos de rendimiento y costos [Dai et al., 2016]. En términos generales, la computación en la nube combina oportunidades como escalabilidad y agilidad con retos como la gestión de costos, la dependencia de proveedores y las preocupaciones de seguridad y cumplimiento [Radapaneni, 2020].

CAPÍTULO 3

PROPUESTA DE SOLUCIÓN

Este capítulo presenta el proceso propuesto para mejorar la eficiencia y la calidad en la entrega de software en equipos pequeños mediante la adopción incremental de buenas prácticas de control de cambios, automatización (CI/CD) e infraestructura/configuración como código (IaC/CaC). El diseño del proceso se deriva de las brechas identificadas en el diagnóstico y de prácticas reportadas en la literatura sobre ingeniería de software continua y DevOps [Fitzgerald and Stol, 2017, Forsgren et al., 2018, Laukkanen et al., 2017].

El proceso se concibe como una receta aplicable de forma incremental en un proyecto pequeño, priorizando prácticas mínimas verificables con evidencia disponible y dejando explícitas las limitaciones cuando una métrica o evidencia no sea observable en el contexto del caso [Forsgren et al., 2018].

3.1. Definiciones operativas y métricas de evaluación

Para asegurar consistencia entre diagnóstico, propuesta y validación, se emplean las siguientes definiciones operativas:

Integración continua (CI). Se entiende como la práctica de integrar cambios de forma frecuente a una rama principal, con validaciones automáticas (por ejemplo, *lint*, formateo, compilación y pruebas) ejecutadas en cada cambio o solicitud de fusión, de manera que los defectos se detecten tempranamente [Fitzgerald and Stol, 2017, Zampetti et al., 2021].

Entrega/Despliegue continuo (CD). Se entiende como la automatización del empaquetado y despliegue de versiones hacia ambientes (por ejemplo, *staging* y producción), incluyendo controles y verificaciones que reduzcan el riesgo del despliegue [Fitzgerald and Stol, 2017, Laukkanen et al., 2017].

Infraestructura como código (IaC) y configuración como código (CaC). Se entiende como la especificación versionada de infraestructura y configuración, de modo que entornos puedan aprovisionarse y configurarse de forma reproducible y auditable, reduciendo cambios manuales [Morris, 2020, Brikman, 2022, Hochstein and Moser, 2017].

Métricas de entrega. Para comparar la situación *pre* y *post* aplicación del proceso se consideran métricas de desempeño comúnmente usadas en DevOps [Forsgren et al., 2018]:

- **Lead time (LT):** tiempo transcurrido entre un cambio listo para integrar (por ejemplo, solicitud de cambio con validaciones completas) y su despliegue en el ambiente objetivo.

- **Frecuencia de despliegue (FD):** número de despliegues efectivos en un periodo definido.
- **Change failure rate (CFR):** proporción de despliegues que introducen una falla y requieren corrección (por ejemplo, *hotfix* o *rollback*).
- **Mean time to restore (MTTR):** tiempo para restaurar el servicio tras una falla asociada a un despliegue.

Dado que en equipos pequeños puede no existir instrumentación suficiente para observar todas las métricas, se adopta el criterio “**no observable**” cuando no exista evidencia verificable (p. ej., ausencia de registros de incidentes). En esos casos se reporta explícitamente la limitación y, cuando sea posible, se emplean *proxies* trazables (por ejemplo, tiempos estimados a partir de marcas de *pipeline*, o recuento de *rollbacks* desde el registro de despliegues).

3.2. Alcance, supuestos y principios de diseño

La propuesta está orientada a proyectos desarrollados por equipos pequeños (por ejemplo, entre 1 y 6 personas), con roles poco especializados y baja madurez en automatización. El proceso no pretende generalizar resultados a la industria completa, su propósito es ofrecer un marco de adopción incremental para contextos comparables a los casos analizados y validarlo mediante un caso piloto.

Se asumen las siguientes condiciones mínimas para aplicar el proceso en un piloto: (i) acceso al repositorio y a su historial, (ii) acceso a ejecutar y observar al menos un *pipeline* de despliegue (CD) o, en su defecto, posibilidad de instrumentarlo y (iii) capacidad de capturar evidencia verificable (IDs de *pipelines*, SHAs/etiquetas de artefacto, registros de cambios y fechas). Si alguno de estos supuestos no se cumple, se documenta como amenaza a la validez y se limita el alcance de la fase correspondiente.

En su diseño se privilegian los siguientes principios: adopción incremental con *quick wins*, automatización como mecanismo para reducir trabajo manual, trazabilidad explícita de cambios y despliegues, reproducibilidad de entornos cuando sea viable; y medición *pre/post* para sustentar resultados [Forsgren et al., 2018, [Laukkanen et al., 2017]].

3.3. Estructura general del proceso

La Tabla 1 sintetiza las seis fases del proceso, sus objetivos y evidencia mínima esperada. La descripción detallada de cada fase se presenta a continuación.

Tabla 1: Resumen del proceso propuesto: fases, objetivo y evidencia mínima.

Fuente: Elaboración propia.

Fase	Objetivo	Evidencia mínima verificable
1. Preparación y línea base	Seleccionar el proyecto, caracterizar el contexto y establecer una línea base de métricas y brechas observables.	Tabla de línea base (LT, FD, CFR, MTTR cuando aplique) y registro de fuentes (repo/pipelines/logs).
2. Control de cambios y calidad	Reducir riesgo de integración mediante políticas de ramas/PR y validaciones automáticas mínimas.	Reglas de rama principal, PRs con <i>checks</i> y evidencia de ejecuciones de CI (o equivalentes).
3. Empaquetado y CD	Definir artefacto desplegable y automatizar despliegue a <i>staging</i> con verificaciones y promoción controlada.	Registro de artefactos (tag/SHA) y ejecuciones de CD con resultados de <i>smoke/health checks</i> .
4. IaC/CaC y entornos	Aumentar reproducibilidad de entornos y reducir cambios manuales mediante IaC/CaC integrados al flujo.	Repositorio de IaC/CaC, evidencia de <i>plan/apply</i> (o equivalente) y control por PR.
5. Operación y trazabilidad	Asegurar identificabilidad y reversibilidad de despliegues con <i>logs</i> y procedimiento de rollback.	Versionado visible por ambiente, <i>logs</i> persistentes y evidencia de un <i>rollback drill</i> (o prueba controlada).
6. Medición y ajuste	Comparar <i>pre/post</i> , analizar brechas remanentes y definir ajustes.	Tabla comparativa <i>pre/post</i> con fuentes, análisis de variaciones y plan de mejora priorizado.

3.4. Trazabilidad entre brechas del diagnóstico y el proceso

La propuesta se justifica a partir de brechas identificadas en el diagnóstico. Para mantener trazabilidad, la Tabla 2 mapea brechas priorizadas hacia las fases del proceso y los artefactos que aportan evidencia de cierre parcial o total.

Tabla 2: Trazabilidad entre brechas del diagnóstico, fases del proceso y evidencia asociada.
Fuente: Elaboración propia.

Brecha priorizada (diagnóstico)	Fase(s) que la aborda(n)	Evidencia de implementación / cierre
Ausencia de estrategia explícita de control de cambios (ramas/PR)	2	Políticas de rama principal, PRs con revisión y <i>checks</i> bloqueantes.
Ausencia o baja cobertura de validaciones automáticas antes de integrar cambios (CI)	2	Ejecuciones de validaciones automáticas (<i>lint/build/tests</i>) en cada PR. Configuración de <i>checks</i> .
Despliegue manual o de alta fricción, con baja repetibilidad	3	<i>Pipeline</i> de CD reproducible con despliegue a <i>staging</i> y verificaciones post-deploy.
Cambios de infraestructura/configuración manuales y propensos a error	4	Repositorio IaC/CaC. Evidencia de <i>plan/apply</i> controlado (o diseño objetivo si no es factible).
Baja trazabilidad de despliegues y ausencia de rollback ejecutable	5	Registro SHA/tag por ambiente, <i>logs</i> persistentes y <i>runbook</i> de rollback con <i>drill</i> .
Dificultad para cuantificar mejoras y sostener cambios	1,6	Tablas <i>pre/post</i> con fuentes y análisis de amenazas a la validez.

3.5. Descripción de las fases del proceso

3.5.1. Fase 1: Preparación y establecimiento de línea base

Propósito. Esta fase establece las condiciones mínimas para evaluar el proceso. Se selecciona un proyecto piloto con control interno del despliegue y se caracteriza su contexto (equipo, stack, ambientes y restricciones). Luego, se define una ventana de observación y se levantan métricas y brechas observables a partir de evidencia disponible.

En equipos pequeños es frecuente que las métricas no estén instrumentadas. Por ello, el levantamiento prioriza evidencia accesible: historial del repositorio (PRs, merges, tags), registros del *pipeline* de CD y, cuando existan, registros de incidentes o *hotfix*. Cuando una métrica no pueda observarse, se reporta como “no observable” y se documenta la causa.

Tabla 3: Fase 1: prácticas mínimas, artefactos y criterios de aceptación.

Fuente: Elaboración propia.

Elemento	Descripción
Prácticas mínimas	Selección del piloto con acceso a repositorio y evidencia de despliegues, definición de métricas y su semántica, definición de ventana de observación (p. ej., 4–6 semanas o N cambios comparables), identificación de fuentes y restricciones, captura de evidencia por despliegue (ID <i>pipeline</i> , SHA/tag, fechas).
Artefactos/evidencia	Ficha de caracterización del piloto, tabla de línea base (LT, FD, CFR, MTTR cuando aplique) con enlaces/IDs, registro de fuentes y limitaciones.
Criterios de aceptación	Piloto caracterizado, métricas definidas con fuentes, línea base completada con evidencia verificable o marcada como “no observable” con justificación.
Puntos de control	Revisión de línea base y riesgos iniciales (ausencia de pruebas, falta de versionado por ambiente, dependencia de terceros).

3.5.2. Fase 2: Control de cambios y calidad mínima

Propósito. Esta fase reduce el riesgo de integrar cambios introduciendo una política explícita de control de cambios (estrategia de ramas, revisión de PR y protección de rama principal) y validaciones automáticas mínimas. La selección de estrategia (por ejemplo, GitHub Flow, Trunk-Based o Gitflow) debe justificarse según el tamaño del equipo, cadencia de liberación y necesidad de soporte [Driessen, 2010, GitHub, Hammant,].

En proyectos donde no existan pruebas automatizadas, el proceso propone iniciar con validaciones de bajo costo (*lint*/formato y compilación) y planificar la incorporación gradual de pruebas, priorizando las rutas críticas de negocio. Este criterio alinea el proceso con la disponibilidad real de artefactos y evita exigir, en la validación, activos de pruebas inexistentes.

Tabla 4: Fase 2: prácticas mínimas, artefactos y criterios de aceptación.

Fuente: Elaboración propia.

Elemento	Descripción
Prácticas mínimas	Definir y documentar estrategia de ramas, proteger <i>main/trunk</i> y exigir PR/MR, establecer <i>checks</i> mínimos (<i>lint/format</i> , <i>build</i> y/o pruebas básicas) ejecutados automáticamente y bloqueantes, definir convención mínima de commits/PR y plantilla de PR.
Artefactos/evidencia	Política de ramas, configuración de protecciones, historial de PRs con <i>checks</i> visibles, plantilla de PR en repositorio.
Criterios de aceptación	Rama principal protegida, PRs obligatorias con <i>checks</i> bloqueantes, evidencia de ejecución automática en PR.
Puntos de control	Definir excepciones acotadas y documentadas, revisión periódica de cumplimiento (muestra de PRs).

3.5.3. Fase 3: Empaquetado y despliegue continuo (CD) con verificaciones

Propósito. Esta fase formaliza el concepto de “versión desplegable” y automatiza el despliegue hacia un ambiente intermedio (*staging*) con verificaciones posteriores. Para reducir variabilidad, se recomienda promover el mismo artefacto entre ambientes, evitando reconstrucciones ad-hoc y mejorando la trazabilidad del despliegue [Laukkanen et al., 2017, Fitzgerald and Stol, 2017].

Dado que los datos disponibles en este trabajo incluyen evidencia directa de *pipelines* de CD, esta fase se considera central para la validación: permite capturar tiempos de despliegue, frecuencia y resultados, además de anclar la trazabilidad a SHAs/tags.

Tabla 5: Fase 3: prácticas mínimas, artefactos y criterios de aceptación.

Fuente: Elaboración propia.

Elemento	Descripción
Prácticas mínimas	Definir artefacto (imagen, paquete o binario) y convención de etiquetado inmutable con SHA, publicar en un registro/repositorio cuando aplique, ejecutar CD automático a <i>staging</i> tras <i>merge</i> , separar variables/secretos por ambiente, incorporar verificaciones <i>smoke/health</i> y criterio de fallo, definir <i>gates</i> para producción (aprobación/policy) y promoción del mismo artefacto.
Artefactos/evidencia	Definición de artefacto y etiquetado, <i>pipeline</i> de CD con registros de ejecuciones, evidencia de verificaciones post-deploy, política de <i>gates</i> y promoción.
Criterios de aceptación	Despliegue automático a <i>staging</i> con verificaciones, artefacto identificable por SHA/tag, <i>gate</i> para producción configurado o documentado (si aplica).
Puntos de control	Auditoría de aprobaciones, registro de fallas y <i>rollbacks</i> (si ocurren).

3.5.4. Fase 4: IaC/CaC y reproducibilidad de entornos

Propósito. Esta fase busca reducir el esfuerzo manual y la variabilidad de entornos mediante IaC/CaC versionados, revisables y, cuando sea viable, integrados al flujo de entrega [Morris, 2020, Brikman, 2022].

Dado que no todos los proyectos pequeños disponen de control total de la infraestructura, la adopción de IaC/CaC se plantea como incremental y condicionada a la factibilidad del piloto (por ejemplo, que el equipo tenga permisos para aprovisionar recursos o al menos automatizar configuración). Si el piloto no permite cambios de infraestructura, esta fase se limita a documentar el diseño objetivo y se explicita como trabajo futuro.

Tabla 6: Fase 4: prácticas mínimas, artefactos y criterios de aceptación.
Fuente: Elaboración propia.

Elemento	Descripción
Prácticas mínimas	Estructurar IaC mínima (módulos y variables por ambiente) con estado remoto, ejecutar <i>validate/plan</i> (o equivalente) en PR y <i>apply</i> con <i>gate</i> , describir configuración idempotente (CaC) cuando aplique, centralizar secretos por ambiente y mínimos privilegios, documentar paridad y diferencias entre entornos.
Artefactos/evidencia	Repositorio IaC/CaC, evidencia de <i>plan</i> revisado, registros de <i>apply</i> (si aplica), documentación de variables/secretos y paridad de entornos.
Criterios de aceptación	Entorno <i>dev/staging</i> reproducible desde código o, si no es factible, diseño objetivo documentado con restricciones.
Puntos de control	Revisión de deriva (<i>drift</i>), auditoría de permisos de despliegue/infraestructura.

3.5.5. Fase 5: Operación, trazabilidad y reversión

Propósito. Esta fase asegura que cada despliegue sea identificable (SHA/tag), auditable (*logs*) y reversible (procedimiento de rollback). En contextos de equipos pequeños, la principal contribución es reducir dependencia de conocimiento tácito mediante *runbooks* mínimos y evidencia de ejecución controlada.

Tabla 7: Fase 5: prácticas mínimas, artefactos y criterios de aceptación.

Fuente: Elaboración propia.

Elemento	Descripción
Prácticas mínimas	Registrar SHA/tag por despliegue y exponer versión por ambiente (endpoint/label), conservar <i>logs</i> de <i>pipeline</i> y, cuando sea posible, de aplicación, definir <i>runbook</i> de rollback con precondiciones y verificación, ejecutar una prueba controlada (<i>rollback drill</i>) en ambiente no productivo cuando sea viable.
Artefactos/evidencia	<i>Runbook</i> de despliegue/rollback, registros de despliegue con SHA/tag, evidencia de retención de <i>logs</i> , reporte del <i>drill</i> con tiempos y resultados.
Criterios de aceptación	Trazabilidad PR-build-deploy disponible o trazabilidad mínima (SHA/tag + <i>pipeline</i> ID) documentada, <i>runbook</i> probado en ambiente seguro.
Puntos de control	Auditoría de trazabilidad (muestra de despliegues), revisión de lecciones aprendidas del <i>drill</i> .

3.5.6. Fase 6: Medición, análisis y ajuste

Propósito. Esta fase compara el desempeño *pre/post* con una ventana equivalente y analiza la reducción de brechas. Dado el tamaño acotado del piloto, el reporte prioriza rangos (mín-máx) y medidas robustas (por ejemplo, mediana) por sobre promedios simples, y explicita amenazas a la validez (cambios de alcance, estacionalidad, disponibilidad del equipo) [Forsgren et al., 2018].

A diferencia de umbrales rígidos, el criterio de éxito se formula en términos de **mejora relativa** respecto de la línea base y de **evidencia de reducción de trabajo manual** y aumento de trazabilidad. Esto se alinea con el objetivo de “reducir la brecha” más que eliminarla completamente.

Tabla 8: Fase 6: prácticas mínimas, artefactos y criterios de aceptación.

Fuente: Elaboración propia.

Elemento	Descripción
Prácticas mínimas	Re-medición de LT, FD, CFR y MTTR cuando sean observables, reporte por rangos y mediana, análisis de variaciones y brechas persistentes, elaboración de plan de mejora priorizado para una iteración posterior.
Artefactos/evidencia	Tabla <i>pre/post</i> con fuentes (IDs/SHAs/ <i>logs</i>), narrativa de resultados y amenazas a la validez, <i>backlog</i> de mejoras priorizado.
Criterios de aceptación	Evidencia de comparación <i>pre/post</i> , conclusiones sustentadas en datos disponibles y limitaciones explícitas, plan de ajuste definido.
Puntos de control	Revisión con stakeholders del piloto, registro de decisiones y acciones siguientes.

3.6. Estrategia de adopción y artefactos de apoyo

La adopción recomendada prioriza las Fases 1 a 3 como núcleo mínimo (línea base, control de cambios y CD con verificaciones), pues son las que típicamente entregan *quick wins* con evidencia verificable en proyectos de baja madurez. La Fase 5 se incorpora en paralelo en su variante mínima (trazabilidad por SHA/tag y *runbook*), ya que fortalece la operación y facilita medir CFR/MTTR cuando existan incidentes. La incorporación de IaC/CaC (Fase 4) se recomienda cuando el piloto permita intervenir infraestructura y cuando la estandarización de entornos sea un problema relevante.

Los artefactos de apoyo (plantilla de PR, checklist de revisión, esquema mínimo de CI/CD, *skeleton* de IaC y checklist de *rollback drill*) se incluyen como anexos editables para no recargar el cuerpo principal del documento.

3.7. Trazabilidad con la validación

La validación del proceso se apoya en la correspondencia entre fases y evidencia: la Fase 1 define la línea base y las fuentes, la Fase 2 aporta evidencia de control de cambios (políticas de ramas, PRs y ejecuciones de validaciones), la Fase 3 produce evidencia operacional directa desde el *pipeline* de CD (despliegues, tiempos y verificaciones), la Fase 4, cuando se aplica, produce evidencia de reproducibilidad mediante IaC/CaC, la Fase 5 completa la trazabilidad del despliegue y prueba reversión, y la Fase 6 consolida el análisis *pre/post* con amenazas a la validez y un plan de mejora.

CAPÍTULO 4

VALIDACIÓN DE LA SOLUCIÓN

Este capítulo presenta la validación del proceso propuesto mediante su aplicación y análisis en dos proyectos reales, anonimizados como **Proyecto A** y **Proyecto B**. La validación se orienta a comprobar que el proceso es **aplicable** en el contexto de equipos pequeños, y que su adopción se asocia a mejoras observables en (i) automatización del despliegue (CD), (ii) trazabilidad entre cambios y despliegues, y (iii) métricas de entrega cuando existe evidencia verificable (por ejemplo, a partir de *pipelines* y registros de incidentes).

A diferencia de evaluaciones de alcance industrial, esta validación se acota al entorno del autor y a la evidencia realmente disponible (repositorios, ejecuciones de *pipelines* de CD, *tickets* y bitácoras) en un proyectos que forman han sido desarrollados para una empresa con un area de tecnología pequeño, en concordancia con los lineamientos del trabajo y con el objetivo de construir una propuesta factible y demostrable.

4.1. Diseño del estudio

La validación se plantea como un **estudio de caso múltiple** compuesto por dos unidades de análisis (Proyecto A y Proyecto B). Ambos casos fueron seleccionados por su representatividad respecto del problema abordado: proyectos pequeños en etapa temprana, con un equipo acotado, libertad para definir *stack* y proceso, y despliegue sobre infraestructura en la nube (AWS). Adicionalmente, se trata de productos internos, lo que reduce el riesgo operativo asociado a la experimentación con prácticas y automatización.

En términos de confidencialidad, se mantiene la anonimización de empresa y clientes, y se restringe la exposición de información sensible (nombres de recursos, credenciales, variables o capturas que permitan identificar el proyecto).

4.1.1. Ventanas de observación y comparabilidad

Ambos proyectos corresponden a productos nuevos en los cuales se planificó desde el inicio el uso de *pipelines* para despliegue continuo (CD) y una estrategia de control de cambios. Por esta razón, no existe una línea base *pre* cuantitativa completa comparable con un estado anterior del mismo proyecto.

En consecuencia, la comparación se estructura en dos niveles:

1. **Comparación operacional acotada (cuando hay evidencia):** se contrasta el esfuerzo

manual y el tiempo operativo de despliegue previo a la automatización completa con los tiempos observados luego de estabilizar el CD. Esto se justifica en tanto el proceso propuesto busca reducir trabajo manual y variabilidad del despliegue.

2. **Comparación cualitativa con el diagnóstico:** se contrasta el estado de los casos con las brechas identificadas en el diagnóstico (capítulo de definición del problema), particularmente aquellas asociadas a documentación, control de cambios, falta de automatización y baja trazabilidad.

Para asegurar comparabilidad entre proyectos en las métricas reportadas, se adopta el siguiente criterio:

- **Métricas temporales (LT):** se calculan a partir de la duración de ejecuciones de CD asociadas a despliegues exitosos. Para evitar sesgos por valores extremos, se reporta **mediana y rango** (mín-máx). En este capítulo se presenta una muestra verificable de $n = 15$ ejecuciones por proyecto, junto con el detalle (*commit* y *timestamp*).
- **Métricas de conteo (FD):** se reporta el número total de despliegues observados en una ventana de tiempo conocida. Para comparación entre proyectos, se normaliza como despliegues/mes (promedio aritmético).

4.2. Caracterización de los casos

La Tabla 9 resume la caracterización de ambos proyectos. En los dos casos el equipo corresponde principalmente a una sola persona que concentra múltiples roles (desarrollo, QA y operaciones), complementada por *stakeholders* que reportan fallas y validan resultados.

Tabla 9: Caracterización de los proyectos utilizados en la validación (anonimizados).
Fuente: Elaboración propia.

Proyecto	Contexto y tipo de sistema	Equipo y forma de trabajo	Stack / infraestructura y despliegue
Proyecto A	Automatización interna para descarga programada de documentos en ventanas periódicas. Servicio tipo <i>backend</i> dentro de una solución basada en microservicios.	1 desarrollador que cumple roles de desarrollo, QA, DevOps y liderazgo técnico. Gestión de trabajo con <i>Linear</i> (Kanban informal, sin tiempos/esfuerzo). 2 <i>stakeholders</i> validan resultados y reportan errores.	Python 3.12 en AWS Lambda como contenedor. Servicios asociados en AWS (p.ej., S3, SQS, DynamoDB, Secrets Manager). Repositorio en <i>GitHub</i> y CD mediante <i>GitHub Actions</i> . Incluye IaC (estado remoto) según el diseño del estudio.
Proyecto B	Automatización interna para digitalización de documentos PDF hacia JSON estandarizado. Servicio tipo <i>backend</i> dentro de una solución basada en microservicios.	1 desarrollador con roles múltiples (desarrollo, QA, DevOps y liderazgo técnico). Gestión de trabajo con <i>Linear</i> (Kanban informal, sin tiempos/esfuerzo). 2 <i>stakeholders</i> validan digitalizaciones y reportan fallas.	Python 3.12 en AWS Lambda (código desplegado directamente). Servicios asociados en AWS (p.ej., S3, SQS). Repositorio en <i>GitHub</i> y CD mediante <i>GitHub Actions</i> . No incluye IaC .

4.3. Alcance de aplicación del proceso

La validación observa la adopción de las fases del proceso descritas en el capítulo de propuesta de solución. Dado el contexto de proyectos nuevos y equipo pequeño, se prioriza el núcleo mínimo de fases que permite evidenciar mejoras operacionales y métricas verificables: (i) control de cambios, (ii) despliegue continuo, y (iii) trazabilidad y reversión.

Tabla 10: Alcance observado por fase del proceso en cada proyecto.

Fuente: Elaboración propia.

Fase	Proyecto A	Proyecto B	Evidencia/observaciones
Fase 2: Control de cambios y calidad	Sí	Sí	Estrategia de ramificación y uso de PR como mecanismo principal de integración. Convención de commits (<i>conventional commits</i>). En Proyecto A existe validación automática bloqueante (<i>lint</i>) previo al merge. En el Proyecto B no se cuenta con este bloqueo. Se observaron excepciones acotadas con commits directos a <i>main</i> por hotfix.
Fase 3: Empaquetado y CD	Sí	Sí	CD automático ante <i>push</i> a <i>main</i> . Proyecto A empaqueta como imagen <i>Docker</i> versionada y publicada en un registro. En el Proyecto B despliega código directamente a Lambda (sin <i>tag</i> de artefacto equivalente).
Fase 4: IaC/CaC y entornos	Sí	No	Proyecto A incluye IaC con estado remoto y permisos acotados a recursos previstos. Proyecto B no incorpora IaC, en su lugar, la infraestructura se mantiene fuera del repositorio. Ambos proyectos operan actualmente con un solo ambiente por tratarse de productos en prueba interna.
Fase 5: Operación, trazabilidad y reversión	Parcial	Parcial	Se mantiene trazabilidad entre <i>commit</i> y despliegue mediante registros de CD. Proyecto A adicionalmente versiona el artefacto con SHA en el <i>tag</i> de imagen. El Proyecto B no registra en la infraestructura (<i>Lambda</i>) el SHA del código desplegado. En ambos casos el <i>rollback</i> puede realizarse mediante <i>git revert</i> y re-despliegue automático.
Fase 6: Medición y ajuste	Sí	Sí	Se dispone de evidencia para medir LT/FD y para registrar fallas y tiempos de recuperación a partir de <i>tickets</i> y despliegues. La línea base <i>pre</i> es limitada por el carácter nuevo de los proyectos, por lo que se reporta como “no disponible” cuando corresponda.

4.3.1. Definiciones operativas adicionales: *runbook* y *rollback drill*

Dado que estos conceptos se utilizan como evidencia recomendada en la fase de operación, se adoptan las siguientes definiciones para el presente estudio:

Runbook de despliegue/rollback. Documento breve y operacional que describe, paso a paso, cómo ejecutar un (des)pliegue y cómo revertirlo. Debe incluir: precondiciones (permisos, credenciales), comandos o acciones en la herramienta de CD, criterios de verificación (*health checks*, *smoke tests*) y criterios de cierre.

Rollback drill. Ejercicio controlado en el cual se ejecuta un *rollback* siguiendo el *runbook* para comprobar que el procedimiento es correcto y que el sistema puede restaurar una versión anterior en un tiempo acotado. Idealmente se realiza en un ambiente no productivo (*staging*)

o en un periodo de bajo riesgo.

4.4. Métricas, fuentes de evidencia y método de cálculo

Las métricas principales se alinean con métricas de desempeño utilizadas en DevOps (LT, FD, CFR, MTTR) [Forsgren et al., 2018]. Adicionalmente, se reporta trazabilidad como un indicador cualitativo/estructural crítico para equipos pequeños.

Tabla 11: Métricas utilizadas, definición operativa y fuente primaria.
Fuente: Elaboración propia.

Métrica	Definición operativa en este estudio	Fuente primaria
Frecuencia de despliegue (FD)	Conteo de ejecuciones exitosas del <i>pipeline</i> de CD que actualizan el ambiente objetivo (en este caso, producción interna). Se reporta como total por ventana y promedio mensual.	Registros de <i>GitHub Actions</i> (<i>runs</i> de CD).
Lead time (LT)	Tiempo entre integración y despliegue. Dado que el CD se ejecuta automáticamente ante <i>push</i> a <i>main</i> , se aproxima como la duración del run de CD hasta la finalización exitosa del despliegue. Se reporta mediana y rango sobre una muestra verificable ($n = 15$).	Registros de <i>GitHub Actions</i> (duración y <i>timestamps</i> por run).
Change failure rate (CFR)	Proporción de despliegues asociados a una falla que requiere corrección. Se emplean definiciones operativas por proyecto (según cómo se identifican los <i>fixes</i> en <i>commits</i> y su relación temporal).	Historial <i>Git</i> (mensajes de <i>commit</i>) + bitácora/ <i>tickets</i> + registros de despliegue.
Mean time to restore (MTTR)	Tiempo desde el reporte/detección del incidente (<i>ticket</i>) hasta que una corrección se encuentra desplegada en el ambiente. Se calcula sobre incidentes documentados con timestamp.	<i>Tickets</i> /bitácora + <i>timestamps</i> de despliegue.
Trazabilidad (TRZ)	Capacidad de asociar un despliegue con un <i>commit</i> SHA y, cuando aplique, con una versión/tag de artefacto. Se evalúa mediante una rúbrica (Tabla 17).	<i>Logs</i> de CD, <i>tags</i> de artefactos (si existen), repositorio.

4.5. Presentación de evidencia y trazas

Con el fin de mantener el cuerpo principal del documento legible y a la vez verificable, la evidencia se reporta de la siguiente manera:

- Se incluye en este capítulo una muestra de ejecuciones de CD por proyecto (Tablas 18 y 19) que permite verificar tiempos, commits y fechas.
- La figura 2 muestra un ejemplo de cómo se relacionan a través del SHA del *commit* el repositorio, los *logs* de CD y el *tag* de artefacto (Proyecto A).
- La figura 3 muestra un ejemplo de ejecución de CD para Proyecto B, evidenciando la ausencia de *tag* de artefacto y la trazabilidad limitada al *commit* en el repositorio y *logs* de CD.

4.5.1. Muestra de despliegues y tiempos observados

Las Tablas 18 y 19 presentan un subconjunto de ejecuciones exitosas ($n = 15$ por proyecto), extraídas de los registros de CD. Estas tablas cumplen dos propósitos: (i) mostrar cómo se presentarán los *logs* en un formato verificable, y (ii) servir como base para el cálculo descriptivo del LT reportado.

4.6. Resultados cuantitativos

4.6.1. Frecuencia de despliegue y lead time

En Proyecto A se registraron **43 despliegues** (ejecuciones de CD) en una ventana de **4 meses**, lo que equivale a un promedio aritmético de **10.75 despliegues/mes**. En Proyecto B se registraron **91 despliegues** en **1 año**, correspondientes a un promedio aritmético de **7.58 despliegues/mes**.

El *lead time* (LT), aproximado mediante la duración del *pipeline* de CD, muestra diferencias marcadas entre proyectos, explicables por la maduración de los proyectos: mientras Proyecto A lleva aproximadamente 4 meses en producción, Proyecto B lleva más de un año.

Tabla 12: Resumen de FD y LT (muestra $n=15$ para LT).

Fuente: Elaboración propia.

Indicador	Proyecto A	Proyecto B
Frecuencia de despliegue (FD)	43 despliegues en 4 meses (aprox. 10.75 despliegues/mes)	91 despliegues en 1 año (aprox. 7.58 despliegues/mes)
Lead time (LT) — mediana (muestra)	117s (1m57s)	15s
Lead time (LT) — rango mín-máx (muestra)	101s-129s (1m41s-2m09s)	11s-28s

4.6.2. Tasa de fallos por cambio y tiempo de recuperación

La tasa de fallos por cambio (CFR) se calculó utilizando definiciones operativas coherentes con la evidencia disponible en cada proyecto.

En Proyecto A se considera “fallo” cuando posterior a un despliegue existe otro despliegue ejecutado por un *commit* identificado como *fix* dentro del mismo día y que modifica el mismo archivo. Bajo esta definición se observaron **3 fallos en 43 despliegues (CFR = 6.98 %)**.

En Proyecto B se consideran fallos aquellos *commits* directos a *main* sin PR, identificados como *fix*, y que interrumpieron el funcionamiento en ciertos casos (excluyendo cambios de formato en la información). Bajo esta definición se observaron **4 fallos en 50 despliegues (CFR = 8.00 %)**.

El tiempo de recuperación (MTTR) se estimó como el tiempo desde el reporte del incidente (*ticket*) hasta que la corrección se encuentra en producción, utilizando timestamps disponibles. Las Tablas 13 y 14 detallan los incidentes reportados.

Tabla 13: Incidentes y MTTR observado — Proyecto A (n=3).

Fuente: Elaboración propia.

Commit que introduce falla	Hora despliegue (falla)	Hora despliegue (fix)	MTTR aprox.
7e148a8	Nov 27, 2025 (17:53)	Nov 27, 2025 (18:04) [9785d1f]	11 min
ffc8334	Nov 5, 2025 (10:40)	Nov 5, 2025 (10:51) [080f4c4]	11 min
Oca4a13	Nov 5, 2025 (16:07)	Nov 5, 2025 (16:18) [7279e89] (+fix adicional a 16:32)	11–25 min

Tabla 14: Incidentes y MTTR observado — Proyecto B (n=4).

Fuente: Elaboración propia.

Fix (commit)	Hora reporte (ticket)	Hora fix en prod	MTTR aprox.
0a622a5	Oct 7, 2025 (09:56)	Oct 7, 2025 (12:36)	2h40m
4d00051	Oct 2, 2025 (14:30)	Oct 2, 2025 (14:43)	13 min
c49e6bd	Oct 14, 2025 (10:12)	Oct 14, 2025 (11:05)	53 min
a0cd589	Oct 29, 2025 (17:42)	Oct 29, 2025 (18:58)	1h16m

Tabla 15: Resumen de CFR y MTTR observado.

Fuente: Elaboración propia.

Indicador	Proyecto A	Proyecto B
CFR	3/43 = 6.98 %	4/50 = 8.00 %
MTTR (muestra) — mediana	11 min	64.5 min
MTTR (muestra) — rango mín-máx	11-25 min	13-160 min

4.6.3. Esfuerzo manual del despliegue

Aunque no existe una línea base cuantitativa completa, ambos proyectos reportan evidencia cualitativa del esfuerzo manual requerido antes de estabilizar el despliegue automatizado.

En Proyecto B, en el inicio del proyecto no existía CD y el despliegue manual tomaba aproximadamente **10 minutos**, además de introducir riesgo de errores manuales (por ejemplo, traspaso incorrecto de archivos o dependencias). Posterior a incorporar CD, los tiempos operativos del despliegue se reducen a las duraciones observadas en los *pipelines* (Tabla 19).

En Proyecto A se registró un episodio donde el pipeline CD no actualizaba la Lambda automáticamente, requiriendo una intervención manual adicional de aproximadamente **5 minutos** posterior a la ejecución del pipeline. Una vez incorporada la actualización de Lambda dentro del pipeline, dicha intervención dejó de ser necesaria.

Tabla 16: Evidencia cualitativa del esfuerzo manual de despliegue (antes/después).

Fuente: Elaboración propia.

Proyecto	Antes (evidencia cualitativa)	Después (evidencia cualitativa + medición)
Proyecto A	Intervención manual adicional posterior al CD (aprox. 5 min) para actualizar Lambda cuando el pipeline no lo incluía.	Despliegue automatizado completo mediante CD. Tiempos observados del CD entre 1m41s y 2m09s (Tabla 12).
Proyecto B	Despliegue manual sin CD (aprox. 10 min) y riesgo de error manual en actualización de archivos.	Despliegue automatizado mediante CD. Tiempos observados entre 11s y 28s (Tabla 12).

4.7. Resultados de trazabilidad

La trazabilidad es un requisito central para interpretar métricas y para facilitar reversión y auditoría. En este estudio se evalúa trazabilidad con una rúbrica simple que refleja la evidencia observable en los proyectos.

Tabla 17: Rúbrica de trazabilidad (TRZ) basada en evidencia verificable.

Fuente: Elaboración propia.

Criterio	Proyecto A	Proyecto B
C1. Existe asociación commit SHA → ejecución de CD (<i>logs</i>)	Sí	Sí
C2. Existe asociación ejecución de CD → versión/artefacto versionado (<i>tag</i>)	Sí (<i>tag</i> incluye SHA)	No (sin <i>tag</i> equivalente)
C3. Es posible reconstruir la versión desplegada a partir de la evidencia (sin acceder a datos sensibles)	Sí (SHA + <i>tag</i> + <i>logs</i>)	Parcial (SHA + <i>logs</i> . Versión en Lambda no expone SHA)
C4. Existe procedimiento explícito de rollback (<i>runbook</i>) y evidencia de prueba controlada (<i>drill</i>)	Parcial (<i>rollback</i> vía <i>git revert</i> . Sin <i>drill</i> documentado)	Parcial (<i>rollback</i> vía <i>git revert</i> . Sin <i>drill</i> documentado)

4.8. Resultados cualitativos

Los resultados cualitativos, levantados desde observaciones del autor y *feedback* de *stakeholders*, se organizan en beneficios y dificultades.

Beneficios observados. En ambos proyectos, la automatización de despliegues redujo trabajo repetitivo y facilitó iteración rápida. En Proyecto A, la identificación explícita de artefactos mediante *tags* basados en SHA facilitó la trazabilidad y simplificó el proceso de revertir a una versión anterior. En Proyecto B, el paso desde despliegue manual hacia CD disminuyó tiempos operativos de despliegue y redujo errores manuales asociados a la actualización de archivos.

Dificultades observadas. En ambos proyectos, el uso estricto de prácticas asociadas a revisión de código por PR se ve condicionado por el tamaño del equipo: al existir un solo desarrollador, la revisión por pares no es sistemática, por lo que el rol de validaciones automáticas (CI) se vuelve más relevante. Adicionalmente, la ausencia de IaC en Proyecto B implica menor trazabilidad de cambios de infraestructura y mayor esfuerzo en modificaciones operativas, particularmente cuando se agregan servicios nuevos.

4.9. Discusión y síntesis multi-caso

Los resultados cuantitativos evidencian que, en ambos casos, el despliegue automatizado es viable y proporciona tiempos acotados de entrega hacia producción interna. El contraste entre proyectos aporta evidencia adicional: Proyecto A, al versionar artefactos (imagen Docker) y contar con IaC, presenta mayor capacidad de trazabilidad técnica y control sobre la infraestructura. El proyecto B, al carecer de versionado explícito de artefacto, tiene des-

ventajas respecto a como los metodos que se pueden seguir para revertir los cambios y al no tener IaC no se tiene conocimiento exacto de la infraestructura que se está utilizando, lo cual afecta la trazabilidad y potencialmente la facilidad de auditoría.

En términos de calidad operativa, ambos proyectos muestran CFR de magnitud similar (aprox. 7–8 %) bajo las definiciones operativas adoptadas. Sin embargo, el MTTR difiere significativamente: Proyecto A presenta tiempos de recuperación en el orden de minutos, mientras que Proyecto B incluye incidentes con recuperación de hasta 2h40m. Este resultado sugiere que, además de automatización de despliegue, el tiempo de restauración está influido por factores adicionales (por ejemplo, detección, priorización, complejidad del defecto, y disponibilidad temporal), los cuales deben ser abordados con instrumentación y procedimientos operativos (por ejemplo, *runbooks* más formales y pruebas controladas de *rollback*).

CAPÍTULO 5

CONCLUSIONES

5.1. Conclusiones respecto del diagnóstico y la brecha inicial

El diagnóstico permitió confirmar que, en equipos pequeños, la madurez de prácticas de entrega no depende únicamente del conocimiento técnico disponible, sino también de restricciones operativas y organizacionales. En los casos estudiados se observó una **brecha recurrente** en tres dimensiones: (i) ausencia o debilidad de integración continua (CI), (ii) despliegue continuo (CD) parcial, manual o externalizado, y (iii) trazabilidad limitada entre versiones, cambios y despliegues.

En particular, el diagnóstico evidenció que la CI era inexistente o marginal en la mayoría de los proyectos analizados, lo que reduce la capacidad del equipo para detectar regresiones de forma temprana. Del mismo modo, aun cuando algunos proyectos disponían de mecanismos de despliegue automatizado, éstos tendían a carecer de elementos complementarios que incrementan confiabilidad y control (por ejemplo, verificación post-despliegue, promoción consistente de artefactos entre ambientes y observabilidad asociada a incidentes). Además, se observó que la trazabilidad (registro explícito del *commit* o versión desplegada por ambiente) era limitada o inexistente, lo que obstaculiza auditoría y reversión rápida ante fallas.

Un hallazgo central del diagnóstico fue que, en contextos donde el despliegue depende de terceros o de aprobaciones externas, el *lead time* tiende a estar dominado por factores no técnicos (gestiones, coordinación y permisos), resultando en tiempos del orden de días en ciertos casos. En contraste, en proyectos donde el despliegue es interno, el *lead time* tiende a estar dominado por tiempos de desarrollo y disponibilidad de revisores. En ambos escenarios, la brecha de automatización y trazabilidad se mantiene como un factor transversal que afecta la calidad y la eficiencia.

Podemos decir que la brecha inicial caracterizada en el diagnóstico justificó la necesidad de una propuesta orientada a: (i) establecer una base mínima de control de cambios y calidad, (ii) sistematizar el despliegue mediante automatización verificable, y (iii) asegurar trazabilidad suficiente para medir, auditar y recuperar el servicio con menor costo.

5.2. Conclusiones respecto del proceso propuesto

El proceso propuesto constituye el aporte central del trabajo: una metodología incremental para mejorar eficiencia y calidad en la entrega de *software* en equipos pequeños, basada en buenas prácticas y sustentada en la brecha diagnosticada. A diferencia de aproximaciones centradas en herramientas específicas, el proceso se definió como una secuencia de fases

con objetivos claros, artefactos mínimos y criterios de aceptación observables, de modo que su adopción no dependa de instrumentación compleja ni de estructuras organizacionales propias de equipos grandes.

Desde una perspectiva metodológica, el proceso aporta dos elementos relevantes. Primero, explicita el principio de **verificabilidad**: cada fase propone prácticas mínimas que pueden sostenerse con evidencia (por ejemplo, *runs* de CD, historial de *commits*, *tickets* de incidentes), lo que permite evaluar resultados incluso cuando no existen tableros formales o mediciones históricas completas. Segundo, incorpora una orientación a **adopción incremental**, lo que reduce el costo inicial de implementación y permite priorizar cierres de brecha según restricciones del proyecto (tiempo, acceso a infraestructura, dependencia de terceros, entre otros).

En términos de alineamiento con el diagnóstico, el proceso prioriza la automatización del despliegue (CD) y la trazabilidad como condiciones habilitantes para iteración rápida y controlada, y considera CI e IaC/CaC como prácticas que incrementan estabilidad y reproducibilidad cuando son factibles para el equipo y el contexto. Este diseño es coherente con la evidencia levantada: en equipos pequeños, el costo de errores manuales y de falta de visibilidad suele ser proporcionalmente más alto, dado que existe menor redundancia de roles y menor capacidad de absorción de incidentes.

5.3. Conclusiones respecto de la validación y comparación con la brecha inicial

La validación del proceso en dos proyectos reales (Proyecto A y Proyecto B) permite concluir que la metodología es **aplicable** en equipos pequeños y que se asocia a mejoras observables en automatización y trazabilidad, utilizando fuentes de evidencia disponibles en entornos prácticos (por ejemplo, *pipelines* de CD y registros de incidentes).

En cuanto a automatización del despliegue, los resultados cuantitativos muestran que ambos proyectos sostienen una frecuencia de despliegue alta para contextos de equipos mínimos: Proyecto A registró 43 despliegues en 4 meses (10.75 despliegues/mes) y Proyecto B registró 91 despliegues en un año (7.58 despliegues/mes), según la Tabla 12. Estos valores contrastan con la situación observada en el diagnóstico, donde la frecuencia de despliegue se aproximaba típicamente al orden semanal en varios casos, y donde el despliegue podía ser manual o externalizado, incrementando tiempos de espera y variabilidad.

Respecto del *lead time*, al aproximarlos mediante la duración del *pipeline* de CD y reportar mediana y rango sobre una muestra verificable ($n = 15$ por proyecto), se observaron tiempos acotados: 117s (1m57s) de mediana en Proyecto A y 15s de mediana en Proyecto B (Tabla 12). En el diagnóstico, se reportaron proyectos donde el *lead time* típico era del orden de días debido a coordinación y restricciones, así como proyectos con tiempos menores a un día en despliegues internos. La evidencia de la validación permite afirmar que, cuando

el despliegue es automatizable y el equipo dispone de control sobre el *pipeline*, el proceso facilita alcanzar tiempos de entrega operativos del orden de segundos o minutos, solo dependiendo del tiempo que demore en construir el artefacto que es publicado, reduciendo fricción y habilitando iteración rápida.

En términos de calidad operativa, fue posible calcular CFR y MTTR a partir de definiciones operativas coherentes con la evidencia disponible. Proyecto A presentó 3 fallos en 43 despliegues (CFR 6.98 %) y Proyecto B presentó 4 fallos en 50 despliegues (CFR 8.00 %), con MTTR mediano de 11 minutos y 64.5 minutos, respectivamente (Tabla 15). En los proyectos del diagnóstico, CFR y MTTR se encontraban mayoritariamente no observables o no registrados, lo que refleja una brecha adicional: la dificultad de medir y aprender de incidentes cuando no existen prácticas mínimas de registro y trazabilidad. En este sentido, más allá de los valores específicos de CFR/MTTR, el resultado relevante es que la adopción del proceso permitió instrumentar métricas con datos disponibles sin requerir plataformas adicionales.

La trazabilidad constituye uno de los contrastes más relevantes respecto de la brecha inicial. En el diagnóstico se observó ausencia de mecanismos claros para asociar despliegues con versiones o *commits* en varios proyectos, lo que obstaculiza auditoría y reversión. En la validación, ambos proyectos presentan asociación *commit* → ejecución de CD (logs), y Proyecto A además implementa versionado explícito del artefacto mediante un tag que incorpora SHA, lo cual permite reconstruir la versión desplegada (Tabla 17). Proyecto B, en cambio, mantiene una limitación de trazabilidad a nivel de infraestructura, al no exponer un identificador de versión en la Lambda desplegada, lo que se refleja en una evaluación parcial del criterio C2/C3. Este contraste demuestra que la trazabilidad no es un atributo binario: depende del tipo de artefacto y de cómo se materializa el despliegue, y requiere decisiones explícitas en el diseño del *pipeline* y de los artefactos de entrega.

Por último, la validación también aporta evidencia cualitativa sobre reducción de esfuerzo manual: se registró una disminución desde despliegues manuales de aproximadamente 10 minutos en Proyecto B, y una intervención manual de 5 minutos en un episodio inicial de Proyecto A, hacia despliegues automatizados con tiempos consistentes, según la Tabla 16. Este resultado se alinea con la brecha inicial, donde el despliegue manual y la dependencia de personas específicas eran factores que incrementaban variabilidad y riesgo.

5.4. Cumplimiento de objetivos

El objetivo general del trabajo: Diseñar un proceso basado en buenas prácticas para automatizar el despliegue y la entrega de *software* en equipos pequeños, se considera cumplido en la medida en que: (i) el proceso se deriva explícitamente de la brecha diagnosticada, (ii) se estructura en fases con criterios verificables, y (iii) se valida su aplicabilidad con evidencia en proyectos reales.

Respecto de los objetivos específicos, se concluye lo siguiente:

- El análisis de brecha se realizó sobre un conjunto acotado de casos, caracterizando prácticas y restricciones.
- El proceso fue diseñado con enfoque incremental y trazabilidad hacia las brechas del diagnóstico.
- El proceso se aplicó en proyectos reales, permitiendo analizar resultados con evidencia verificable y métricas operativas.
- La evaluación de resultados se efectuó utilizando métricas de entrega y trazabilidad cuando fueron observables, complementando con comparación cualitativa con el diagnóstico cuando no existió una línea base pre cuantitativa completa.

5.5. Aportes y contribuciones

A partir del trabajo realizado, se identifican las siguientes contribuciones:

1. **Caracterización de la brecha en un contexto acotado:** el diagnóstico documenta brechas recurrentes en automatización, trazabilidad y medición en equipos pequeños, sustentadas con evidencia disponible.
2. **Proceso incremental basado en buenas prácticas:** se propone una metodología estructurada por fases, con prácticas mínimas verificables, orientada a ser aplicable bajo restricciones típicas de equipos pequeños.
3. **Validación empírica con evidencia verificable:** se demuestra aplicabilidad del proceso en dos proyectos, reportando métricas operativas y criterios de trazabilidad con soporte en *pipelines* y registros de incidentes.

5.6. Limitaciones

Las conclusiones deben interpretarse considerando limitaciones inherentes al diseño del estudio. Primero, la validación se realizó sobre dos proyectos y en un contexto organizacional acotado, por lo que la generalización a otros dominios o a equipos con estructuras distintas debe abordarse con cuidado. Segundo, ambos proyectos corresponden a productos nuevos, por lo que no existió una línea base pre cuantitativa completa, además la comparación se apoyó en evidencia operacional acotada (esfuerzo manual) y en contraste cualitativo con el diagnóstico. Tercero, algunas métricas (por ejemplo, CFR y MTTR) dependen de definiciones operativas y de la completitud del registro de incidentes, lo que puede introducir sesgos si existe sub-reportaje. Finalmente, la ausencia de múltiples ambientes y de *runbooks/drills* formales limita la evaluación de prácticas avanzadas (promoción de artefactos entre entornos, *rollback* controlado), las cuales se identifican como oportunidades de extensión.

5.7. Líneas de trabajo futuro

A partir de los resultados obtenidos, se identifican oportunidades de extensión orientadas a robustecer la evidencia y a ampliar aplicabilidad del proceso. Entre ellas, destacan:

1. Ampliar la validación a más proyectos y dominios, con ventanas temporales equivalentes que habiliten comparaciones cuantitativas más directas
2. Formalizar prácticas de operación, tales como *runbooks* de despliegue y ejercicios controlados de reversión, para evaluar de manera más completa la capacidad de recuperación
3. Incrementar la cobertura de CI (pruebas automatizadas y análisis de calidad/seguridad) en proyectos donde actualmente es parcial, de modo de cerrar brechas observadas en el diagnóstico y persistentes en la validación.

ANEXOS

Tabla 18: Muestra de ejecuciones de CD para Proyecto A (n=15).

Fuente: Elaboración propia.

Commit (SHA)	Duración CD	Fecha/hora fin	Nota de evidencia
dca2897	2m05s	Jan 20 (15:40)	Registro de run disponible en logs de CD.
36d0140	2m06s	Jan 13 (18:24)	Registro de run disponible en logs de CD.
0f2326f	2m09s	Dec 11, 2025 (15:00)	Registro de run disponible en logs de CD.
9785d1f	1m55s	Nov 27, 2025 (18:04)	Commit corresponde a fix (incidente).
8b32774	1m57s	Nov 27, 2025 (16:00)	Registro de run disponible en logs de CD.
283b9b8	2m02s	Nov 10, 2025 (18:17)	Registro de run disponible en logs de CD.
19454f5	1m52s	Nov 10, 2025 (18:10)	Registro de run disponible en logs de CD.
a0eda1b	1m48s	Nov 5, 2025 (19:40)	Registro de run disponible en logs de CD.
0ca4a13	1m51s	Nov 5, 2025 (16:07)	Commit asociado a incidente (ver Tabla 13).
b896ffa	2m06s	Nov 5, 2025 (13:11)	Registro de run disponible en logs de CD.
b03cfb6	2m02s	Nov 5, 2025 (12:27)	Registro de run disponible en logs de CD.
cc1a28e	1m57s	Nov 5, 2025 (11:56)	Registro de run disponible en logs de CD.
0d7a6e1	2m05s	Nov 4, 2025 (11:38)	Registro de run disponible en logs de CD.
839294d	1m52s	Nov 3, 2025 (21:41)	Registro de run disponible en logs de CD.
f2a9cf2	1m41s	Nov 3, 2025 (21:07)	Registro de run disponible en logs de CD.

Tabla 19: Muestra de ejecuciones de CD para Proyecto B (n=15).

Fuente: Elaboración propia.

Commit (SHA)	Duración CD	Fecha/hora fin	Nota de evidencia
49dda97	14s	Jan 14 (14:31)	Registro de run disponible en logs de CD.
00c705f	15s	Jan 2 (17:50)	Registro de run disponible en logs de CD.
c564867	15s	Dec 31, 2025 (12:18)	Registro de run disponible en logs de CD.
bd656f0	13s	Dec 22, 2025 (14:04)	Registro de run disponible en logs de CD.
1b19ed1	19s	Oct 9, 2025 (11:30)	Registro de run disponible en logs de CD.
0a622a5	17s	Oct 7, 2025 (12:36)	Commit asociado a incidente (ver Tabla 14).
23d06df	28s	Oct 1, 2025 (18:04)	Registro de run disponible en logs de CD.
fc8ef23	18s	Sep 10, 2025 (14:11)	Registro de run disponible en logs de CD.
0fcc912	15s	Sep 9, 2025 (23:30)	Registro de run disponible en logs de CD.
7fc5180	17s	Sep 4, 2025 (16:17)	Registro de run disponible en logs de CD.
33d9493	11s	Sep 4, 2025 (16:17)	Registro de run disponible en logs de CD.
b47510c	15s	Aug 19, 2025 (16:03)	Registro de run disponible en logs de CD.
d1774be	19s	Aug 19, 2025 (13:10)	Registro de run disponible en logs de CD.
f873391	13s	Aug 19, 2025 (12:38)	Registro de run disponible en logs de CD.
6cfeb2b	24s	Jun 16, 2025 (11:07)	Registro de run disponible en logs de CD.

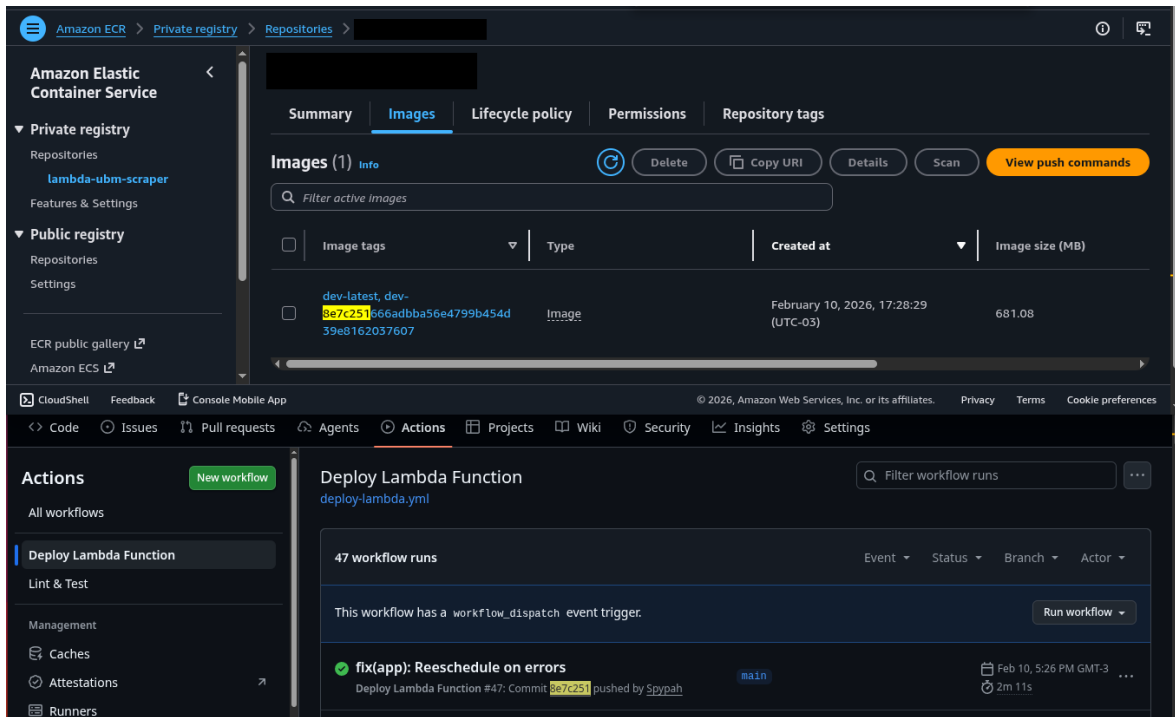


Figura 2: Relación SHA (8e7c251) entre ejecución Pipeline y tag imagen docker
Fuente: Elaboración propia

REFERENCIAS BIBLIOGRÁFICAS

- [Ali et al., 2015] Ali, M., Khan, S. U., and Vasilakos, A. V. (2015). Security in cloud computing: Opportunities and challenges. *Information sciences*, 305:357–383.
- [Astels, 2003] Astels, D. (2003). *Test driven development: A practical guide*. Prentice Hall Professional Technical Reference.
- [Babar and Pantić, 2012] Babar, M. A. and Pantić, Z. (2012). Guidelines for building a private cloud infrastructure. *ITU Technical Report*.
- [Bass et al., 2015] Bass, L., Weber, I., and Zhu, L. (2015). *DevOps: A software architect's perspective*. Addison-Wesley Professional.
- [Beck, 2002] Beck, K. (2002). *Test driven development: By example*. Addison-Wesley Professional.
- [Brikman, 2022] Brikman, Y. (2022). *Terraform: up and running: writing infrastructure as code*. O'Reilly Media, Inc..
- [Buyya et al., 2011] Buyya, R., Broberg, J., Buyya, R., Goscinski, A., Goñi, A., Goñi, A., Goñi, A., Goñi, A., and Goscinski, A. M. (2011). *Cloud computing*. Wiley Online Library.

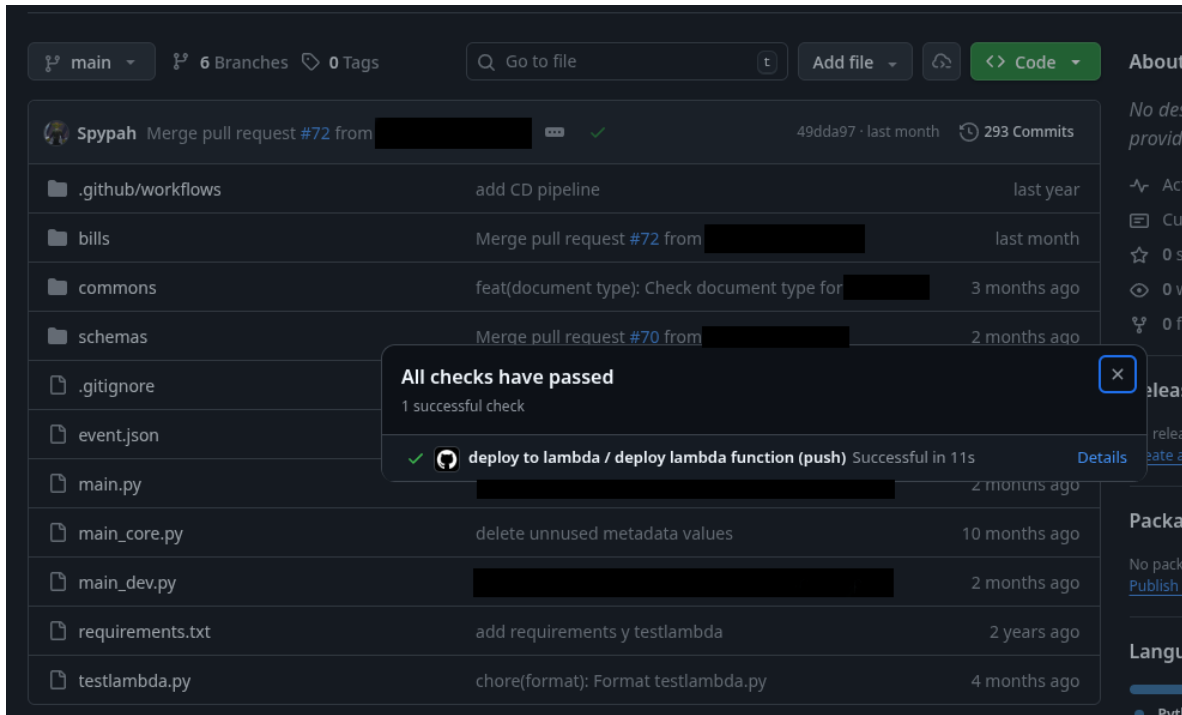


Figura 3: Resultado de ejecución del pipeline CD sobre el *commit* más reciente en el proyecto B

Fuente: Elaboración propia

[Chacon and Straub, 2014] Chacon, S. and Straub, B. (2014). *Pro git*. Springer Nature.

[Claps et al., 2015] Claps, G. G., Berntsson Svensson, R., and Aurum, A. (2015). On the journey to continuous deployment: Technical and social challenges along the way. *Information and Software Technology*, 57:21–31.

[Cowell et al., 2023] Cowell, C., Lotz, N., and Timberlake, C. (2023). *Automating DevOps with GitLab CI/CD Pipelines: Build efficient CI/CD pipelines to verify, secure, and deploy your code using real-life examples*. Packt Publishing Ltd.

[Dai et al., 2016] Dai, W., Qiu, L., Wu, A., and Qiu, M. (2016). Cloud infrastructure resource allocation for big data applications. *IEEE Transactions on Big Data*, 4(3):313–324.

[Dingsøyr et al., 2012] Dingsøyr, T., Nerur, S., Balijepally, V., Moe, N. B., et al. (2012). A decade of agile methodologies: Towards explaining agile software development. *Journal of systems and software*, 85(6):1213.

[Driessen, 2010] Driessen, V. (2010). A successful git branching model. Blog post. Accedido: 2025-11-02.

[Ebert et al., 2016] Ebert, C., Gallardo, G., Hernantes, J., and Serrano, N. (2016). Devops. *IEEE software*, 33(3):94–100.

- [Erdogmus et al., 2005] Erdogmus, H., Morisio, M., and Torchiano, M. (2005). On the effectiveness of the test-first approach to programming. *IEEE Transactions on software Engineering*, 31(3):226–237.
- [Fitzgerald and Stol, 2017] Fitzgerald, B. and Stol, K.-J. (2017). Continuous software engineering: A roadmap and agenda. *Journal of Systems and Software*, 123:176–189.
- [Forsgren et al., 2018] Forsgren, N., Humble, J., and Kim, G. (2018). *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations*. IT Revolution.
- [Fowler et al., 2001] Fowler, M., Highsmith, J., et al. (2001). The agile manifesto. *Software development*, 9(8):28–35.
- [Freeman and Pryce, 2009] Freeman, S. and Pryce, N. (2009). *Growing object-oriented software, guided by tests*. Pearson Education.
- [GitHub,] GitHub. Github flow. Documentation. Accedido: 2025-11-02.
- [GitLab,] GitLab. Gitlab flow. Documentation. Accedido: 2025-11-02.
- [Hammant,] Hammant, P. Trunk-based development. Website. Accedido: 2025-11-02.
- [Hochstein and Moser, 2017] Hochstein, L. and Moser, R. (2017). *Ansible: Up and Running: Automating configuration management and deployment the easy way*. .ºReilly Media, Inc.”.
- [Humble and Farley, 2010] Humble, J. and Farley, D. (2010). *Continuous delivery: reliable software releases through build, test, and deployment automation*. Pearson Education.
- [Janzen and Saiedian, 2005] Janzen, D. and Saiedian, H. (2005). Test-driven development concepts, taxonomy, and future direction. *Computer*, 38(9):43–50.
- [Jayaram et al., 2024] Jayaram, V., Sankiti, S. R., Sughaturu Krishnappa, M., Veerapaneni, P. K., and Carimireddy, P. K. (2024). Accelerated cloud infrastructure development using terraform. V. Jayaram, SR Sankiti, MS Krishnappa, PK Veerapaneni, and PK Carimireddy, *Accelerated Cloud Infrastructure Development Using Terraform, International Journal of Emerging Technologies and Innovative Research*, 11(9):f382–f387.
- [Laukkanen et al., 2017] Laukkanen, E., Itkonen, J., and Lassenius, C. (2017). Problems, causes and solutions when adopting continuous delivery—a systematic literature review. *Information and Software Technology*, 82:55–79.
- [Leppänen et al., 2015] Leppänen, M., Mäkinen, S., Pagels, M., Eloranta, V.-P., Itkonen, J., Mäntylä, M. V., and Männistö, T. (2015). The highways and country roads to continuous deployment. *IEEE Software*, 32(2):64–72.

- [Malla, 2024] Malla, P. (2024). Analyzing the impact of agile methodologies on software quality and delivery speed: A comparative study. *World Journal of Advanced Research and Reviews*.
- [Meszaros, 2007] Meszaros, G. (2007). *xUnit test patterns: Refactoring test code*. Pearson Education.
- [Meyer, 2014] Meyer, M. (2014). Continuous integration and its tools. *IEEE software*, 31(3):14–16.
- [Morris, 2016] Morris, K. (2016). *Infrastructure as code: managing servers in the cloud*. O'Reilly Media, Inc."
- [Morris, 2020] Morris, K. (2020). *Infrastructure as code*. O'Reilly Media.
- [Nerur et al., 2005] Nerur, S., Mahapatra, R., and Mangalaraj, G. (2005). Challenges of migrating to agile methodologies. *Communications of the ACM*, 48(5):72–78.
- [Perez-Riverol et al., 2016] Perez-Riverol, Y., Gatto, L., Wang, R., Sachsenberg, T., Uszkoreit, J., Leprevost, F. d. V., Fufezan, C., Ternent, T., Eglen, S. J., Katz, D. S., et al. (2016). Ten simple rules for taking advantage of git and github.
- [Rafique and Mišić, 2012] Rafique, Y. and Mišić, V. B. (2012). The effects of test-driven development on external quality and productivity: A meta-analysis. *IEEE Transactions on Software Engineering*, 39(6):835–856.
- [Ram, 2013] Ram, K. (2013). Git can facilitate greater reproducibility and increased transparency in science. *Source code for biology and medicine*, 8:1–8.
- [Shahin et al., 2017] Shahin, M., Ali Babar, M., and Zhu, L. (2017). Continuous integration, delivery and deployment: A systematic review on approaches, tools, challenges and practices. *IEEE Access*, 5:3909–3943.
- [Tadapaneni, 2020] Tadapaneni, N. R. (2020). *Cloud computing: Opportunities and challenges*. SSRN.
- [Veragua Albornoz, 2023] Veragua Albornoz, A. A. (2023). Generación de enlaces de pago utilizando un chatbot en whatsapp: Inteligencia artificial aplicada en la industria de telecomunicaciones. pages 11–17.
- [Zampetti et al., 2021] Zampetti, F., Geremia, S., Bavota, G., and Di Penta, M. (2021). Ci/cd pipelines evolution and restructuring: A qualitative and quantitative study. In *2021 IEEE International Conference on Software Maintenance and Evolution (ICSME)*, pages 471–482. IEEE.