



UNIVERSIDAD TÉCNICA
FEDERICO SANTA MARÍA

DEPARTAMENTO DE ELECTROTECNIA E INFORMÁTICA
INGENIERÍA EN INFORMÁTICA

Automatización del desarrollo y despliegue mediante un pipeline CI/CD en plataforma web y móvil para gestión y disponibilidad de planes de emergencia

Maximiliano San Pedro Pino
maximiliano.pino.13@sansano.usm.cl

Pamela Gatica Caballero
Profesora Guía

Franco Jorquera Pezoa
Profesor Correferente

Resumen: FastPlan es una solución web y móvil orientada a agilizar la entrega de información crítica a bomberos durante una emergencia. Habitualmente, los planes de emergencia se mantienen en formatos físicos o extensos (p. ej., PDF), lo que dificulta acceder oportunamente a datos relevantes para la toma de decisiones en ruta, como ubicación de grifos, redes húmedas y secas, tableros eléctricos y zonas con químicos peligrosos.

Esta tesina propone e implementa un pipeline Continuous Integration con generación de artefactos para automatizar la integración del desarrollo y la verificación de compilación de una plataforma compuesta por un frontend web (React + TypeScript), un backend (Node.js + Express) con base de datos MySQL y una aplicación móvil (Flutter), incorporando servicios de apoyo para sincronización/comunicación.

La metodología aplica prácticas DevOps en un equipo de seis integrantes, utilizando GitHub Actions para ejecutar checkout, instalación de dependencias, análisis estático/lint, pruebas básicas (smoke tests) y build; adicionalmente se verifica la construcción de una imagen Docker y, para móvil, la generación de un APK en modo debug. Como resultado, se obtiene una verificación automática y reproducible ante cambios (push/pull request), reduciendo la integración tardía y dejando una base escalable para incorporar despliegue a entornos de staging/producción en una etapa posterior.

Palabras clave: DevOps, CI/CD, Continuous Integration, GitHub Actions, Flutter, React.



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

Automatización del desarrollo y despliegue mediante un pipeline CI/CD en plataforma web y móvil

Título del trabajo: para gestión y disponibilidad de planes de emergencia

Nombre del candidato(a): Maximiliano Andres Pino San Pedro

Carrera / Grado: Ingeniería en Informática

Campus: Viña del Mar Departamento: Electrotecnia e Informática

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Pamela Alexandra Gatica Caballero, 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 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 una 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: 27/10/2026 Firma:

Estudiante o Candidato(a):

Fecha: 27/10/2026 Firma:

Este formulario debe ser insertado como página 2 de la memoria o tesis, completado y firmado por estudiante y profesor(a) antes de la entrega en portal PRISMA de Biblioteca USM.

Introducción

1.1 Contexto y problemática

Las emergencias estructurales, y en particular los incendios, requieren que los equipos de primera respuesta tomen decisiones en pocos minutos y con información incompleta. En ese escenario, existe un tipo de información que resulta crítica para la operación: la ubicación de elementos como grifos, redes húmedas, redes secas, tableros eléctricos, accesos, zonas de seguridad y presencia de químicos peligrosos, entre otros. En la práctica, esta información suele estar disponible en formatos físicos, en documentos extensos (por ejemplo, planos y archivos PDF), o distribuida en distintas fuentes, lo que dificulta consultarla con rapidez y precisión durante el desplazamiento hacia la emergencia.

Bajo este contexto se desarrolla FastPlan, una plataforma compuesta por una aplicación web y una aplicación móvil interconectadas, cuyo objetivo es facilitar la disponibilidad de información crítica de un plan de emergencia de manera rápida y visual. La aplicación web permite gestionar y enviar información desde un punto central (por ejemplo, un operador o encargado), y la aplicación móvil está orientada a ser consumida por bomberos en ruta hacia una emergencia, mostrando en un mapa la ubicación de elementos críticos mediante íconos.

1.2 Motivación

El desarrollo de un sistema web y móvil con integración entre múltiples componentes, realizado por un equipo de seis personas, introduce desafíos que van más allá de la funcionalidad del producto: integración frecuente de cambios, gestión de versiones, control de calidad, detección temprana de errores y estandarización de procesos de construcción y pruebas. En proyectos de este tipo, una parte importante de los problemas aparece al final, cuando se intenta unir el trabajo de varias personas o cuando se compila un componente en un entorno distinto al local de desarrollo.

La motivación de esta tesina surge precisamente de esa necesidad: reducir la fricción de integración y aumentar la confiabilidad del proceso de construcción y validación, mediante la automatización de actividades repetitivas y propensas a errores. En un producto destinado a escenarios críticos, donde la disponibilidad y estabilidad son relevantes, resulta especialmente importante establecer prácticas que permitan mantener calidad de manera consistente y que, además, permitan escalar el proyecto en una etapa posterior fuera del contexto universitario.

En este sentido, la automatización mediante un enfoque DevOps y un pipeline CI/CD (Integración Continua / Entrega o Despliegue Continuo) se vuelve un eje central: permite validar cada cambio que se integra al repositorio, ejecutar controles de calidad mínimos (linting, pruebas básicas, build), y generar artefactos reproducibles (por ejemplo, builds o imágenes) que sirven como base para incorporar despliegues a futuro.

1.3 Enfoque de la tesina

Es importante diferenciar el alcance del proyecto de título del alcance de esta tesina. Mientras el proyecto FastPlan aborda el desarrollo del sistema (web + backend + móvil)

para gestionar y visualizar planes de emergencia, esta tesina se centra en el diseño, selección, adaptación e implementación de un modelo de automatización CI/CD que soporte el desarrollo de FastPlan y que, a la vez, sienta bases para una evolución escalable del producto.

Para ello, se trabaja sobre una arquitectura tecnológica compuesta por:

- Frontend Web: React + TypeScript
- Backend: Node.js + Express
- Base de datos: MySQL
- Aplicación móvil: Flutter.
- Servicios complementarios: Integración de servicios de Firebase para comunicar las 2 aplicaciones

El pipeline CI/CD se implementa utilizando GitHub Actions, incorporando etapas de verificación automática por componente (web, backend, móvil) y tareas orientadas a estandarizar el proceso de construcción. Dado el estado actual del proyecto, el enfoque se centra en Integración Continua (CI) y generación/verificación de artefactos, dejando explícito como limitación que el despliegue productivo (CD completo a producción) no se ejecutó en infraestructura cloud durante esta etapa, aunque se deja la propuesta preparada para extenderse.

1.4 Planteamiento del problema

En proyectos colaborativos con múltiples componentes, la ausencia de un proceso automatizado de integración y validación tiende a generar errores detectados tardíamente (en integración manual o en “la máquina de otro”), inconsistencias de dependencias y configuración, tiempos elevados para validar cambios y dificultades para establecer estándares mínimos de calidad.

Por lo tanto, el problema que guía esta tesina puede formularse como:

¿Cómo diseñar e implementar un pipeline CI que automatice la integración, validación y construcción de una plataforma web y móvil, desarrollada colaborativamente, con el objetivo de reducir errores de integración, estandarizar el flujo de trabajo y habilitar una futura escalabilidad del proyecto FastPlan?

1.5 Objetivo general

Proponer, adaptar e implementar un pipeline Continuous Integration con generación de artefactos para automatizar el proceso de integración, validación y construcción de la plataforma FastPlan (web, backend y móvil), estableciendo una base reproducible y escalable para su evolución.

1.6 Objetivos específicos:

1. Analizar las necesidades del proyecto FastPlan y del equipo de desarrollo, identificando requerimientos técnicos y operativos para la automatización (repositorios, ramas, dependencias, build y validación).

2. Seleccionar y adaptar un modelo de pipeline CI utilizado en la industria, ajustándolo al contexto de una plataforma multi componente (React/TypeScript, Node/Express, MySQL y Flutter).
3. Implementar workflows de CI en GitHub Actions para frontend, backend y móvil, incorporando un conjunto base de verificaciones de calidad: análisis estático/linting, pruebas de tipo smoke test y verificación de build.
4. Generar artefactos reproducibles (por ejemplo, compilaciones o imágenes) que estandaricen el resultado del proceso y permitan extender el flujo hacia despliegues en etapas posteriores.
5. Evaluar el impacto del pipeline propuesto mediante una métrica objetiva de eficiencia: el tiempo de validación por cambio, comparando la ejecución manual (creación de contenedor y despliegue local) frente a la validación automatizada del pipeline, complementada con el tiempo de ejecución del pipeline y la frecuencia de fallas en integración, identificando mejoras y trabajo futuro.

1.7 Alcance y limitaciones

El alcance de esta tesina considera:

- Automatización del proceso de Integración Continua para los componentes web, backend y móvil.
- Inclusión de validaciones mínimas de calidad (linting y pruebas básicas tipo smoke test) y verificación de construcción (build).
- Uso de contenedores y ejecución local mediante Docker como base de estandarización del entorno.
- Como limitaciones principales:
 - No se implementa un despliegue completo a infraestructura productiva en la nube durante el periodo del proyecto (CD a producción), aunque el diseño queda planteado para habilitarlo en una etapa posterior.
 - La cobertura de pruebas es inicial y limitada; se incorpora como punto de mejora y expansión futura del pipeline (más pruebas unitarias, integración y E2E (end-to-end)).

1.8 Metodología de trabajo

El desarrollo del proyecto FastPlan se realizó por un equipo de seis integrantes, organizado en sprints de dos a tres semanas, con daily entre dos y tres veces por semana, sprint planning al inicio de cada sprint y una retrospectiva al cierre de este. Yo asumí los roles de Scrum Master y DevOps, lo que en la práctica significó coordinar y facilitar el proceso de desarrollo del equipo, además de ser el responsable de proponer e implementar los mecanismos de automatización que son el objeto de esta tesina, cabe decir que el primer sprint no salió como esperábamos y tuvimos que trabajar muchas horas extra para sacar el proyecto adelante, lo que retrasó que definiéramos hábitos de equipo desde el día uno, algo que fuimos corrigiendo recién en los sprints siguientes.

La metodología propia de esta tesina fue, en el fondo, un proceso de diseño e implementación incremental del pipeline CI/CD, que iba de la mano de los ciclos de trabajo del equipo: en cada sprint aparecía una necesidad concreta de automatización (por ejemplo, verificar el build de un componente o implementar un nuevo control de calidad), se diseñaba e implementaba el workflow correspondiente, se validaba con el equipo durante el sprint, y se ajustaba según lo que salía en la retrospectiva. Este

enfoque incremental me permitió priorizar en cada iteración los mecanismos de automatización con mayor impacto inmediato para el equipo, y evaluar si realmente funcionaban antes de extender el pipeline a nuevos componentes o etapas.

Durante el proyecto y antes de la implementación del pipeline, montar y probar los cambios hechos tomaba alrededor de 25 min, esto es la suma desde iniciar el Docker hasta el debug de la APK, nunca lo cronometramos per se, pero este era el tiempo promedio que nos tomaba ensamblar todo el proyecto.

En resumen, la metodología de la tesina combina gestión ágil (Scrum) para organizar el trabajo en entregas incrementales y coordinar la integración entre integrantes, prácticas DevOps para estandarizar el flujo de integración (repositorios, convenciones de ramas, pull requests y automatizaciones), y la implementación técnica del pipeline mediante GitHub Actions, incorporando tareas de validación y construcción por componente.

1.9 Organización del documento

El documento se estructura de la siguiente forma:

En el Capítulo 2 se presenta el marco teórico relacionado con DevOps, CI/CD, automatización de builds, estrategias de ramas y pruebas, además de herramientas relevantes (GitHub Actions, contenedores, etc.).

En el Capítulo 3 se revisan modelos y prácticas utilizadas en la industria para pipelines CI/CD, estableciendo un punto de comparación y base metodológica para la propuesta.

En el Capítulo 4 se describe el sistema FastPlan desde la perspectiva necesaria para el pipeline: componentes, dependencias, flujo general y requerimientos de automatización.

En el Capítulo 5 se presentan los resultados y la evaluación del pipeline implementado, detallando su diseño por componente (frontend, backend, móvil y Docker), las decisiones de adaptación al contexto del proyecto y las oportunidades de mejora identificadas.

En el Capítulo 6 se propone una hoja de ruta para la evolución del pipeline y las prácticas DevOps, alineada con las tres etapas de crecimiento definidas para FastPlan.

Finalmente, en el Capítulo 7 se exponen las conclusiones, limitaciones y líneas de trabajo futuro para extender la automatización hacia despliegues y mayor cobertura de pruebas.

2 Marco teórico

Este capítulo presenta los conceptos fundamentales que sustentan la tesina, abordando DevOps, CI/CD, control de versiones, automatización con GitHub Actions, contenedores con Docker y estrategias de aseguramiento de calidad mediante pruebas. El propósito es establecer un lenguaje común y un marco conceptual que permita comprender el diseño e implementación del pipeline aplicado posteriormente al caso FastPlan.

2.1 DevOps: colaboración y automatización en el ciclo de vida del software

DevOps es un enfoque de trabajo que integra prácticas y cultura de colaboración entre desarrollo (*Dev*) y operaciones (*Ops*), con el objetivo de mejorar la entrega de software mediante automatización, retroalimentación continua y estandarización de procesos [1]. En lugar de tratar el despliegue y la operación como etapas separadas y tardías, promueve que la calidad, la integración, la seguridad y la mantenibilidad estén presentes desde el inicio del desarrollo.

A nivel práctico, esto se materializa en:

- Automatización de tareas repetitivas (build, pruebas, empaquetado, publicación).
- Estandarización del flujo de trabajo (ramas, revisiones, reglas de integración).
- Observabilidad y retroalimentación (logs, monitoreo, métricas), para detectar y corregir fallas temprano.
- Entrega incremental, reduciendo cambios grandes y riesgosos por cambios pequeños y verificables.

En proyectos con múltiples componentes (frontend, backend, base de datos, móvil), este enfoque ayuda a reducir la fricción entre equipos y asegura que cada cambio pase por verificaciones mínimas antes de integrarse.

2.2 Integración Continua (CI)

La Integración Continua (*Continuous Integration*, CI) es una práctica donde los cambios de código se integran frecuentemente al repositorio principal, y cada integración activa un proceso automatizado que verifica que el sistema [2]:

1. Compila/construye correctamente
2. Cumple estándares (lint, formato, análisis estático)
3. Pasa pruebas automatizadas (al menos básicas)

La idea central de CI es detectar errores lo antes posible, evitando que el equipo descubra fallas recién al final del ciclo, por ejemplo, al juntar ramas o preparar una entrega. De esta forma, se reduce la "deuda de integración" y se entregan señales rápidas ("*feedback loop*") a los desarrolladores.

Elementos típicos de CI:

- Triggers: eventos que disparan el pipeline (push, pull request, tags).
- Jobs/Steps: unidades de ejecución (instalar dependencias, ejecutar tests, build).
- Quality gates: condiciones para aprobar (por ejemplo: tests OK, lint OK, build OK).
- Artefactos: resultados del pipeline (binarios, APK, bundles web, imágenes Docker).

2.3 Entrega Continua y Despliegue Continuo (CD)

Dentro de CD se distinguen dos conceptos:

- Entrega Continua (*Continuous Delivery*): el software queda en un estado “listo para desplegar” en todo momento. El pipeline puede generar artefactos versionados y listos, pero el paso a producción puede requerir una decisión manual o aprobación [3], [2].
- Despliegue Continuo (*Continuous Deployment*): extensión de lo anterior donde, si el cambio pasa las verificaciones, se despliega automáticamente a un entorno objetivo (staging o producción).

En el contexto de una tesina universitaria, y especialmente si no existe infraestructura productiva establecida, es común implementar CI de forma completa y dejar el CD como despliegue a entornos de prueba internos, o bien como preparación técnica para un despliegue futuro (artefactos, contenedores, configuración).

Lo clave, incluso sin despliegue a producción, es que el pipeline entregue reproducibilidad: el build que corre en CI representa un estándar del proyecto.

2.4 Pipeline CI/CD como modelo de proceso

Un pipeline CI/CD es la secuencia automatizada que transforma cambios de código fuente en resultados verificables (builds, pruebas, artefactos y eventualmente despliegues) [2]. Puede verse como una cadena de etapas que típicamente incluye:

1. Checkout / obtención del código
2. Instalación de dependencias
3. Análisis estático y linting
4. Pruebas automatizadas
5. Build/compilación
6. Empaquetado y artefactos
7. Despliegue (si existe)
8. Monitoreo y retroalimentación (si existe)

En sistemas compuestos (web + backend + móvil), el pipeline suele dividirse por componente, pero manteniendo una lógica común: cada componente debe ser verificable por separado y, cuando aplica, integrable con los otros.

2.5 Control de versiones con Git y estrategias de ramas

Git es un sistema de control de versiones distribuido que permite registrar cambios, trabajar en paralelo y administrar integración. En equipos, la estrategia de ramas define cómo se propone, revisa e integra el trabajo.

Conceptos relevantes:

- Rama principal (main/master): referencia estable del proyecto.
- Ramas de características (feature branches): cambios acotados por funcionalidad.
- Pull Request (PR): mecanismo para revisar y discutir cambios antes de integrar.
- Merge: integración de cambios al destino (por ejemplo, main o dev).
- Tags: marcadores de versión (v1.0.0, v1.1.0), útiles para releases.

Entre las estrategias más comunes están el trunk-based development, que consiste en cambios pequeños y frecuentes sobre una rama principal, apoyado por CI estricta [4]; Git Flow, que define ramas como develop, release y hotfix, además de main [5], y que resulta más estructurado, aunque puede aumentar la complejidad; y el feature branch workflow, que combina una rama principal con ramas por feature y PR obligatorio.

En CI, estas estrategias se conectan con reglas como:

- Ejecutar el pipeline en PR para impedir la integración de cambios que rompen el build o los tests.
- Ejecutar el pipeline en main como verificación final.
- Ejecutar pipelines especiales al crear tags/releases (por ejemplo, generar artefactos firmados).

2.6 Aseguramiento de calidad: análisis estático, linting y formateo

Antes de ejecutar las pruebas, una práctica esencial es aplicar verificaciones rápidas y baratas, como el linting, que corresponde a reglas de estilo y buenas prácticas, por ejemplo, ESLint en JavaScript/TypeScript, o el análisis de código en Dart/Flutter [6]; el formateo, que asegura consistencia de formato mediante herramientas como Prettier o dart format [7]; y el type checking, o verificación de tipos en TypeScript [8], útil para detectar errores antes de runtime.

Estas herramientas actúan como una forma de "calidad preventiva", ya que reducen defectos simples y mantienen la consistencia dentro de un equipo de múltiples desarrolladores.

2.7 Pruebas de software en CI: tipos y propósito

El pipeline CI/CD se apoya en pruebas automatizadas para detectar fallas temprano. En esta tesina resultan relevantes los siguientes tipos de prueba.

2.7.1 Pruebas unitarias

Validan funciones o unidades pequeñas de código. Son rápidas, aisladas y útiles para lógica de negocio.

2.7.2 Pruebas de integración

Validan la interacción entre componentes, por ejemplo, un endpoint que accede a la base de datos. Suelen ser más lentas y requieren dependencias como bases de datos o servicios externos.

2.7.3 Pruebas end-to-end (E2E)

Simulan flujos completos desde la perspectiva del usuario, tanto en web como en móvil. Son más costosas y sensibles a cambios, pero altamente representativas.

2.7.4 2.7.4 Smoke tests

Son pruebas mínimas para verificar que el entorno "está vivo": que el proyecto compila, que el runner funciona y que un conjunto básico de aserciones pasa. Resultan útiles como primer paso cuando la cobertura de pruebas aún es baja [1].

En un proyecto académico en evolución, una estrategia práctica es comenzar con lint/typecheck, smoke tests y build, para luego ir incrementando la cobertura con pruebas unitarias, de integración y E2E.

2.8 Automatización con GitHub Actions

GitHub Actions es una plataforma de automatización integrada a GitHub que permite definir flujos de CI/CD a través de archivos YAML ubicados generalmente en `.github/workflows` [9].

Entre los conceptos principales se encuentran el workflow, que es el archivo YAML que define cuándo y cómo se ejecuta la automatización; los event triggers, eventos como push, pull_request, workflow_dispatch (manual), release o tag; los jobs, conjuntos de pasos que se ejecutan en un runner (por ejemplo, Ubuntu); los steps, comandos o acciones como instalar Node, ejecutar tests o hacer build; las actions, componentes reutilizables como actions/checkout; los artifacts, archivos generados por el workflow (por ejemplo, builds o reportes); y los secrets, variables sensibles como tokens o claves, almacenadas en GitHub [10].

Entre las ventajas de este enfoque en un proyecto de equipo destacan un estándar único de verificación, donde si pasa CI el cambio se considera integrable; la trazabilidad, mediante el registro de ejecuciones, fallas y tiempos; y la reproducibilidad, ya que todos siguen el mismo proceso independientemente de su equipo local.

2.9 Contenedores y entornos reproducibles con Docker

Docker permite empaquetar una aplicación y su entorno (dependencias, runtime, configuraciones) dentro de una imagen en un contenedor [11]. En CI/CD, Docker se usa principalmente para homogeneizar entornos y evitar el clásico "en mi máquina funciona", estandarizar builds mediante la generación de imágenes versionadas, y facilitar los despliegues al permitir ejecutar contenedores en los entornos destino.

Entre los conceptos relevantes se encuentran el Dockerfile, que es la receta para construir la imagen; la imagen, una plantilla inmutable y *versionable*; el contenedor, que corresponde a una instancia en ejecución de una imagen; y el registry, el repositorio donde se almacenan las imágenes (Docker Hub, GitHub Container Registry).

Incluso si el proyecto no se despliega en la nube, construir la imagen en CI es valioso para verificar consistencia y preparar una etapa posterior.

2.10 Artefactos, versionado y trazabilidad

Un objetivo central del pipeline es producir artefactos: resultados empaquetados, compilados o listos para consumo, como el build web, el binario del backend, el APK debug o la imagen Docker.

Entre las buenas prácticas asociadas se encuentran:

- Versionar artefactos (tags semánticos o número de build) [12].
- Trazabilidad: asociar artefactos a commits, pull requests y releases.
- Separación por entornos: builds de desarrollo vs builds de release.

En plataformas móviles, el versionado toma especial importancia porque los binarios se distribuyen de forma distinta a la web, considerando firmas, publicación en tiendas y release builds. Por ello, la tesina considera al menos la generación y verificación de builds como base del proceso.

2.11 Gestión de configuración y secretos

Los sistemas web y móvil suelen depender de variables de entorno como URLs, claves o tokens. Frente a esto, un pipeline correcto debe evitar incluir secretos en el repositorio, usar los secrets del proveedor (GitHub Secrets) para CI, diferenciar las configuraciones por entorno (dev/staging/prod), y mantener consistencia, de modo que el pipeline falle si falta alguna configuración crítica.

Esto reduce los riesgos de seguridad y evita errores de integración por configuraciones locales inconsistentes.

2.12 Observabilidad como extensión del pipeline

Aunque no siempre se implementa en etapas iniciales, el ciclo DevOps ideal incluye mecanismos de observabilidad [13], como logs centralizados, métricas de rendimiento, monitoreo de disponibilidad y alertas ante fallas.

En la práctica, la observabilidad suele implementarse una vez que existe un entorno de despliegue persistente, ya sea staging o producción. Sin embargo, es parte del marco teórico porque completa el circuito de retroalimentación: entregar software no es el final, sino el inicio de su operación real.

3 Estado del arte

Este capítulo revisa modelos, prácticas y herramientas utilizadas en la industria para automatizar la integración y entrega de software mediante CI/CD, con énfasis en escenarios multiplataforma (web, backend y móvil). El objetivo es identificar enfoques consolidados que sirvan como base para el diseño del pipeline aplicado posteriormente al caso FastPlan, y reconocer tendencias relevantes en calidad, trazabilidad y medición del desempeño de entrega.

3.1 Modelos de referencia para pipelines CI/CD

En la literatura y práctica industrial, el pipeline CI/CD se entiende como una cadena de etapas que transforma cambios de código en un producto verificable. Uno de los enfoques más difundidos es el concepto de *deployment pipeline*, propuesto como una secuencia que incrementa la confianza mediante validaciones progresivas (desde compilación y pruebas rápidas hasta verificaciones más exigentes) y que habilita la automatización de la entrega. Esta idea es promovida ampliamente por el enfoque de *Continuous Delivery*, donde se plantea que construir un pipeline permite ejecutar actividades de verificación de manera continua, “incorporando calidad desde el inicio” del proceso de entrega. [3]

A nivel conceptual, los modelos de referencia suelen organizarse en las siguientes etapas:

- Commit/CI stage: integración frecuente, build y pruebas rápidas.
- Quality stage: pruebas más completas (unitarias, integración, seguridad, performance).
- Artifact stage: empaquetado y versionado de resultados (binarios, bundles, imágenes).
- Release/Deploy stage: despliegue a ambientes (staging/producción), con o sin aprobaciones.

En proyectos que aún no cuentan con infraestructura productiva o políticas de despliegue consolidadas, es común implementar con mayor profundidad las primeras etapas, es decir, CI y artefactos, dejando el despliegue continuo como una evolución natural del pipeline.

3.2 Herramientas CI/CD utilizadas en la industria

En la práctica, los pipelines CI/CD se implementan mediante plataformas que ejecutan workflows automatizados en respuesta a eventos del repositorio (push, pull request, tags, releases). Entre las soluciones de uso extendido se encuentran GitHub Actions, GitLab CI/CD, Jenkins, CircleCI y Azure Pipelines, entre otras.

En particular, GitHub Actions se ha consolidado por su integración nativa con repositorios GitHub y por ofrecer automatización declarativa mediante YAML. La documentación oficial define un *workflow* como un proceso configurable compuesto por uno o más *jobs*, definido en un archivo YAML dentro del repositorio [14]. Esta aproximación facilita estandarizar validaciones por componente (web, backend, móvil) y aplicar reglas homogéneas al equipo, especialmente cuando el desarrollo se organiza en pull requests.

Además, las plataformas modernas incorporan mecanismos para manejo seguro de configuración sensible. En GitHub Actions, por ejemplo, se promueve el uso de Secrets a nivel de repositorio/entorno/organización para variables sensibles utilizadas por los workflows (tokens, claves, credenciales) [8].

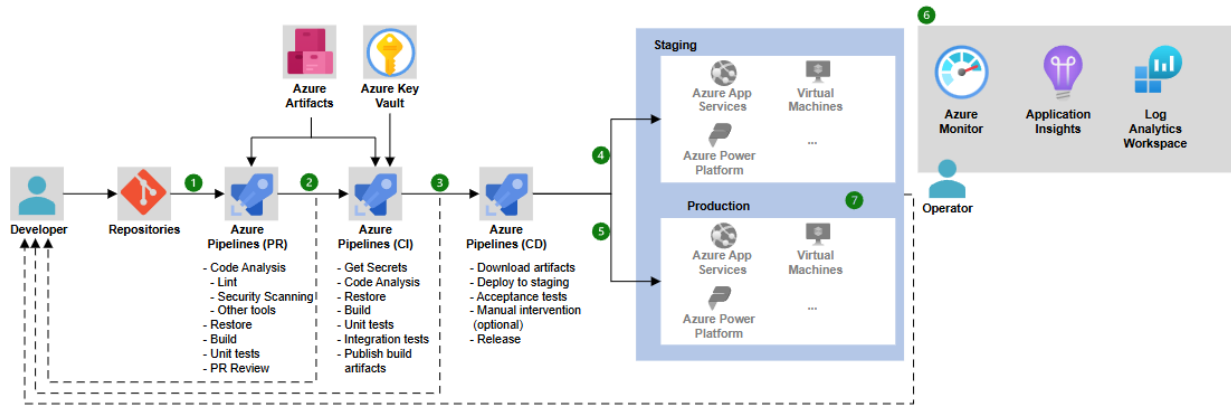


Figura 1. Arquitectura Estándar de Azure
Fuente: Azure DevOps [15]

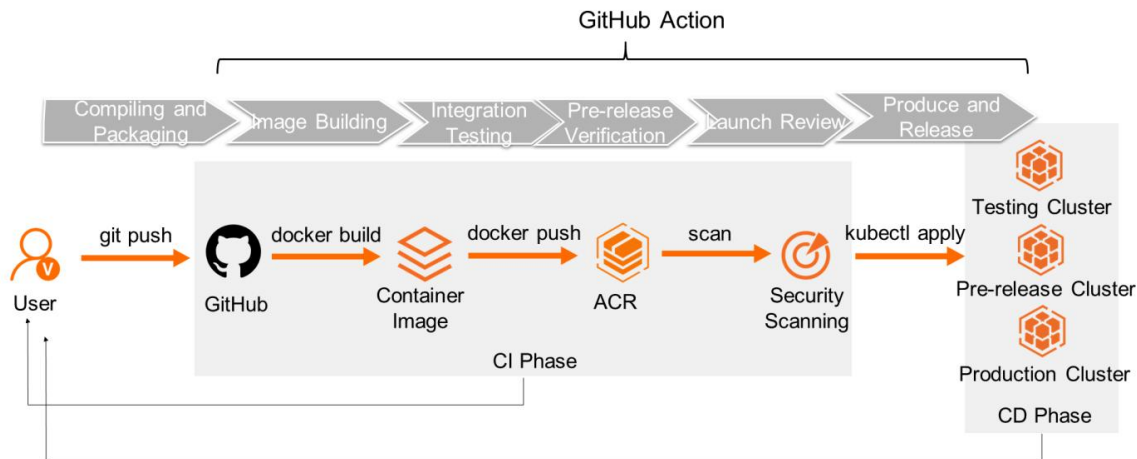


Figura 2. Proceso DevOps basado en GitHub Actions
Fuente: Alibaba Cloud [16]

3.3 Estrategias de ramas y su relación con CI/CD

El estado del arte en CI/CD no se limita a herramientas, sino también a estrategias de integración. En los equipos, la estrategia de ramas define la frecuencia de integración y el riesgo de divergencia entre el trabajo de sus integrantes.

Entre los enfoques más utilizados se encuentra el *trunk-based development* o integración frecuente, que consiste en cambios pequeños integrados frecuentemente a una rama principal, respaldados por CI estricta [4]; el esquema de *feature branches* con Pull Requests, donde cada rama corresponde a una funcionalidad y pasa por revisión y validación previa al merge; y Git Flow, un modelo más estructurado que define ramas

como *develop*, *release* y *hotfix*, útil en ciertos contextos aunque puede aumentar la complejidad y los tiempos de integración si no se gestiona cuidadosamente [5].

En todos los casos, la práctica transversal recomendada por la industria es que el pipeline se ejecute al menos en los pull requests, actuando como quality gate antes de integrar cambios a las ramas estables. Esto reduce la integración tardía y mejora la consistencia del código que llega a la rama principal.

3.4 “Quality gates” en pipelines modernos

Un elemento clave del estado del arte es la incorporación de *quality gates*: condiciones mínimas que el cambio debe cumplir para ser integrable o liberable [17]. En productos web y backend, es común incluir la instalación reproducible de dependencias mediante *lockfiles* y cachés, el análisis estático (linting, formateo y verificación de tipos), pruebas automatizadas que van desde unitarias e integración hasta, progresivamente, E2E, y el build o compilación como verificación determinística de integridad.

En proyectos en etapa temprana o con baja cobertura de pruebas, se suele iniciar con smoke tests y validación de build, incrementando la cobertura con el tiempo. Esta evolución gradua es consistente con el enfoque industrial de aumentar la confianza por etapas, evitando que el pipeline se convierta en un “bloqueo” irrealista para el equipo.

3.5 CI/CD en aplicaciones móviles (desafíos y soluciones actuales)

La automatización en móvil presenta diferencias relevantes frente a web/backend. El estado del arte muestra desafíos adicionales, como la compilación específica por plataforma (Android o iOS) con su respectiva variabilidad de entornos, el manejo de firmas (*code signing*) y credenciales asociadas a la distribución, la generación de artefactos en formatos APK, AAB o IPA junto con sus canales de distribución (testeo interno, tiendas), y un mayor costo de ejecución, dado que los builds suelen ser más lentos y en algunos casos requieren emuladores o simuladores.

Por lo mismo, la industria utiliza herramientas y servicios especializados para CI/CD móvil, especialmente para Flutter/React Native. Plataformas como Codemagic, por ejemplo, enfatizan soporte para build, versionado, firma y publicación en flujos configurables (vía YAML) para proyectos Flutter, incluyendo la gestión de *code signing* como parte del proceso [18].

En contextos donde la publicación a stores no es parte del alcance inmediato, una práctica común (y valiosa) es automatizar al menos:

- ejecución de pruebas (aunque sean mínimas),
- verificación de compilación,
- generación de builds de desarrollo (ej. APK debug), dejando la firma y publicación como extensión posterior.

3.6 Medición del desempeño DevOps: métricas DORA y trazabilidad

El estado del arte contemporáneo en DevOps no solo recomienda automatizar, sino medir. Las métricas DORA (*Four Keys*) [19], [20] se han popularizado como indicadores

de rendimiento de entrega: *deployment frequency*, *lead time for changes*, *change failure rate* y *time to restore service*, que equilibran velocidad la frecuencia de despliegue y lead time con confiabilidad operacional (*change failure rate* y *time to restore service*)

En escenarios donde aún no hay despliegue a producción, estas métricas pueden adaptarse usando proxies observables en CI, como el tiempo de ejecución del pipeline, la tasa de fallas en PR, la consistencia del build y la frecuencia de integración exitosa.

4 Contexto del sistema y requerimientos del pipeline CI/CD

Este capítulo describe el sistema FastPlan desde la perspectiva necesaria para diseñar e implementar un pipeline CI/CD: su composición por componentes, responsabilidades técnicas, flujo general de comunicación y las condiciones reales de trabajo del equipo (repositorios, ramas, herramientas y restricciones). El propósito es levantar los requerimientos que el pipeline debe cubrir para entregar integración reproducible y verificación automática en una plataforma web, backend y móvil.

4.1 Arquitectura técnica y responsabilidades por componente

El sistema se implementó con una arquitectura cliente-servidor, distribuida en los siguientes componentes principales:

- Frontend web (React + TypeScript): interfaz para gestión de planes, carga/edición de información y ejecución de acciones de envío hacia la aplicación móvil. Al ser un cliente web, su proceso de construcción depende de Node.js y del ecosistema de herramientas asociado a Vite/TypeScript (instalación de dependencias, build, linting, pruebas).
- Backend (Node.js + Express): expone servicios para la operación del sistema (APIs), concentra lógica de negocio y actúa como intermediario entre el frontend, la base de datos y los mecanismos de comunicación/sincronización. Su proceso de verificación requiere instalación de dependencias, ejecución de linter (si existe) y pruebas (si están definidas), además de garantizar que el servidor sea construible/ejecutable de forma reproducible.
- Base de datos (MySQL): persistencia de información estructurada del sistema (entidades del plan, usuarios/roles según corresponda, y/o datos operativos). Para el pipeline, esto introduce el desafío clásico de diferenciar: verificación del código (CI) vs. pruebas de integración que requieren servicios externos (DB); dado el alcance actual, la CI se centra en verificación de build y pruebas unitarias/básicas.
- Aplicación móvil (Flutter): cliente para consumo de información en ruta, con énfasis en visualización cartográfica y lectura rápida. Su proceso de verificación en CI implica instalar Flutter en el runner, obtener dependencias (`flutter pub get`), ejecutar análisis estático (`flutter analyze`), ejecutar pruebas (`flutter test`) y verificar compilación mediante build de Android (por ejemplo APK debug).
- Servicios complementarios (Firebase): se utiliza como soporte para comunicación/sincronización entre componentes (por ejemplo, mensajería, autenticación o intercambio de datos, según configuración final del proyecto). Desde la perspectiva del pipeline, esto suele traducirse en manejo de

configuración por ambiente (claves, endpoints, archivos de configuración), y en la necesidad de mantener secretos fuera del repositorio.

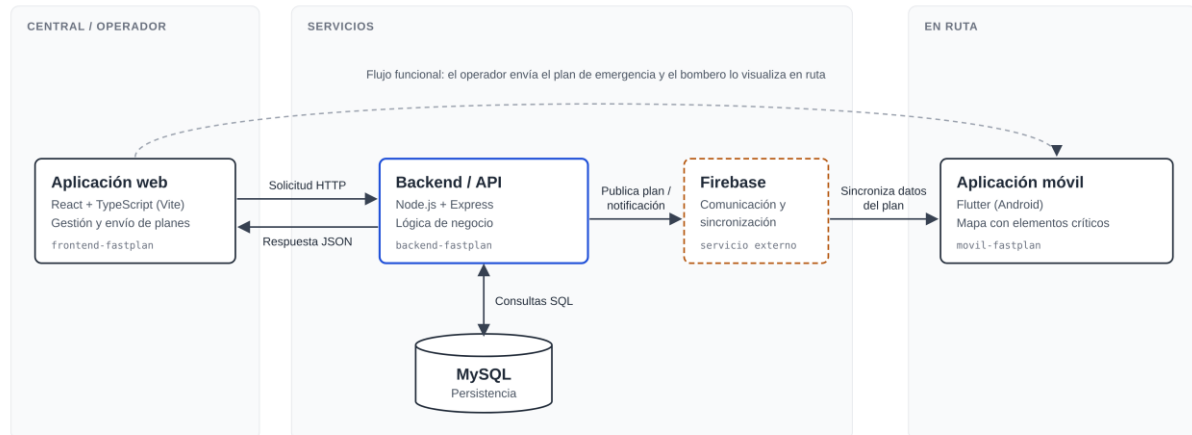


Figura 3. Arquitectura de FastPlan y flujo entre componentes.

Fuente: elaboración propia.

4.2 Organización del código: repositorios y separación de responsabilidades

El desarrollo se organizó en múltiples repositorios, separando el trabajo por componente para facilitar el paralelismo del equipo y reducir acoplamiento de cambios:

1. Repositorio frontend-fastplan (web).
2. Repositorio backend-fastplan (API/servidor).
3. Repositorio movil-fastplan (Flutter).

Esta decisión habilita independencia en el ciclo de construcción de cada componente, pero introduce un requisito directo para CI/CD: cada repositorio debe contar con su propio pipeline, ya que un cambio en el frontend no necesariamente coincide temporalmente con un cambio en el backend o en el móvil, y cada uno tiene dependencias y herramientas de compilación distintas.

4.3 Flujo de trabajo colaborativo y estrategia de ramas

El equipo estuvo compuesto por seis personas y el trabajo se organizó de manera colaborativa, donde cada integrante mantenía ramas propias para desarrollar funcionalidades o realizar cambios aislados, integrando posteriormente hacia ramas de referencia.

En el repositorio web, por ejemplo, se utilizaron ramas como main (principal), dev, ramas de tipo feature y release, además de ramas de respaldo y recuperación. En el

repositorio backend se observó un patrón similar, con main, dev, ramas individuales y ramas de prueba. En móvil, en cambio, se evidencian ramas orientadas a revertir o restaurar estabilidad, junto con ramas por integrante.

En términos de CI/CD, este esquema impone que el pipeline se ejecute al menos sobre la rama principal para asegurar una línea estable, sobre una rama de pruebas/integración (por ejemplo, Test), para validar la integración temprana, y en eventos de Pull Request, actuando como quality gate previo a la integración.

4.4 Entorno de ejecución y práctica de despliegue durante el proyecto

Durante el desarrollo, el sistema se ejecutó principalmente en entorno local, apoyándose en Docker para estandarizar la ejecución y facilitar el levantamiento de servicios. No se realizó un despliegue productivo a infraestructura en la nube como parte del alcance del semestre.

Esta condición define el alcance real del pipeline implementado:

- Se prioriza Integración Continua (CI): verificar que el código compila, que las dependencias están correctas y que pasan validaciones mínimas.
- Se prepara base para Entrega/Despliegue continuo (CD) en una etapa posterior, mediante generación de artefactos (por ejemplo, build web, imagen Docker, APK debug), sin automatizar aún la publicación a un entorno permanente.

4.5 Requerimientos del pipeline CI/CD

A partir del contexto anterior, el pipeline propuesto debe cumplir los siguientes requerimientos técnicos y operativos:

4.5.1 4.5.1 Requerimientos transversales (para todos los repositorios)

1. Ejecución automática por eventos: al menos en push y pull_request hacia ramas objetivo (por ejemplo, main y una rama de pruebas/integración).
2. Reproducibilidad: instalación determinística de dependencias (uso de npm ci cuando exista package-lock.json, y flutter pub get en móvil).
3. Verificación mínima de calidad: incorporar linting/análisis estático cuando esté disponible, sin impedir la ejecución inicial del pipeline en caso de ausencia de scripts (en etapas tempranas).
4. Pruebas automatizadas mínimas: ejecutar al menos pruebas tipo smoke test para confirmar que el entorno funciona y que el runner puede ejecutar tests.
5. Build como verificación de integridad: confirmar que el proyecto es construible en un entorno limpio (runner), reduciendo el riesgo de dependencias implícitas locales.
6. Trazabilidad de fallos: toda falla debe quedar registrada en el sistema de CI (logs y estado de ejecución), de modo que el equipo pueda detectar rápidamente qué cambio rompió la verificación.

4.5.2 4.5.2 Requerimientos específicos por componente

Frontend (React + TypeScript)

El workflow de frontend comienza instalando las dependencias con Node.js, para luego ejecutar el linting cuando el script está disponible (npm run lint) y el build de producción (npm run build) como verificación de que el proyecto compila sin errores. A esto se suma la ejecución de pruebas: en el proyecto se incorporaron pruebas mínimas con un framework de testing, en este caso Vitest, orientadas a un smoke test que confirma que el entorno de build y el bundle generado están operativos.

Backend (Node + Express)

En el caso del backend, el pipeline instala las dependencias mediante npm install o npm ci, según exista o no un lockfile, y ejecuta el linting cuando el script correspondiente está definido. Las pruebas se ejecutan cuando el proyecto las contempla; en su ausencia, el paso se mantiene como una verificación "no bloqueante" en esta primera etapa, dejando abierta su evolución hacia una validación obligatoria conforme aumente la cobertura. Adicionalmente, se mantiene una versión fija de Node (18 LTS) a lo largo del pipeline, lo que asegura reproducibilidad entre los entornos de los distintos integrantes del equipo.

Móvil (Flutter)

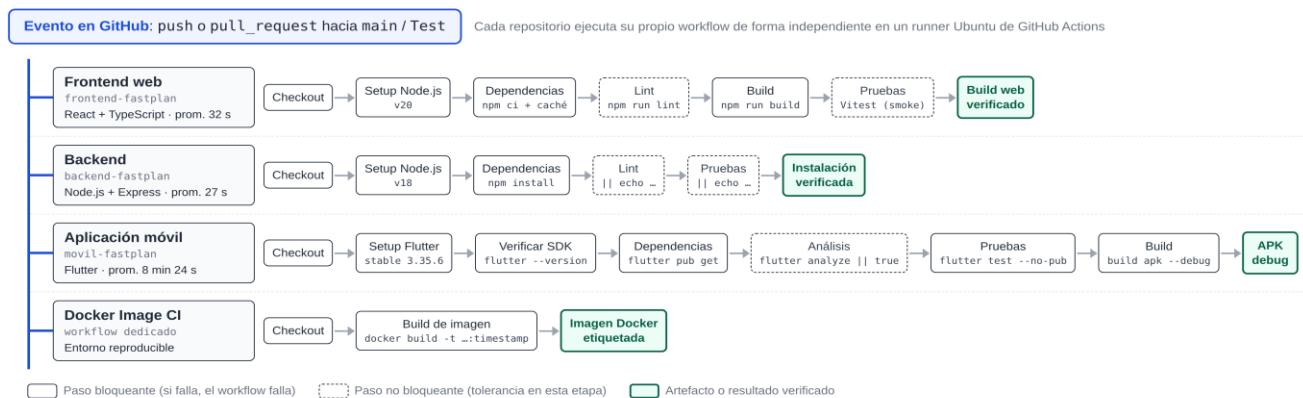


Figura 4 Flujo de Github Actions

Fuente: Elaboración propia.

Como se observa en la Figura 4, el pipeline móvil instala Flutter en el runner, fijando el canal estable y una versión definida para evitar variaciones entre ejecuciones, y luego obtiene las dependencias del proyecto. Posteriormente se ejecuta el análisis estático (flutter analyze), configurado en esta etapa inicial para no bloquear el pipeline ante advertencias, y las pruebas (flutter test), que incluyen un smoke test destinado a verificar que el entorno de pruebas está correctamente configurado. Como paso final, se

construye un APK en modo debug (flutter build apk --debug), lo que permite confirmar que el proyecto compila de extremo a extremo para la plataforma Android.

Docker (entorno reproducible)

Finalmente, el pipeline construye la imagen Docker directamente en CI, verificando que el Dockerfile es válido y que el proyecto puede empaquetarse de forma consistente, independiente del entorno local de cada desarrollador. La imagen se versiona al menos con un identificador de build, lo que sienta las bases para una futura publicación en un registry sin necesidad de rediseñar el proceso.

4.6 Restricciones y supuestos del trabajo

En función del estado y alcance del proyecto, se declaran las siguientes restricciones: no se incorpora automatización de despliegue a staging o producción durante esta etapa, dado que el objetivo inmediato es CI y generación de artefactos; la cobertura de pruebas es limitada, con smoke tests mínimos y una ampliación hacia pruebas unitarias, de integración y E2E que queda como trabajo futuro; y existen dependencias externas como la base de datos y servicios complementarios, que requieren configuración y secretos, los cuales deben administrarse como secretos del repositorio y nunca exponerse en el código.

4.7 Porque Github Actions

Tabla 1. Comparación de plataformas de Integración Continua

Criterio	GitHub Actions	GitLab CI/CD	Jenkins	CircleCI	Azure Pipelines
Integración con los repositorios existentes (GitHub)	Nativa	Requiere migrar o espejar los repositorios	Vía webhooks y plugins	Servicio externo conectado	Servicio externo (Azure DevOps)
Infraestructura que mantener	Ninguna (runners alojados)	Ninguna (SaaS)	Servidor propio, plugins y actualizaciones	Ninguna	Ninguna
Configuración	YAML dentro del repositorio	YAML dentro del repositorio	Jenkinsfile e interfaz web	YAML	YAML
Soporte para Node.js, Flutter y Docker	Actions oficiales y del marketplace	Sí	Sí, mediante plugins	Sí	Sí

Criterio	GitHub Actions	GitLab CI/CD	Jenkins	CircleCI	Azure Pipelines
Costo para el equipo	Plan gratuito suficiente	Plan gratuito	Gratuito, con costo de hosting	Plan gratuito por créditos	Plan gratuito
Curva de aprendizaje y cuentas adicionales	Baja, sin cuentas nuevas	Media	Alta	Media	Media-alta

Fuente: elaboración propia.

Como se observa en la tabla 1, Github Actions fue la opción optima porque ya teníamos todo el proyecto en Github, además de las ventajas que se evidencian en la tabla.

5 Resultados y evaluación del pipeline CI/CD

Este capítulo presenta los resultados obtenidos tras la implementación del pipeline CI/CD en FastPlan, evaluando su funcionamiento en condiciones reales de trabajo del equipo. La evaluación se centra en la capacidad del pipeline para detectar problemas tempranamente, estandarizar la verificación de cambios y garantizar reproducibilidad de builds en un proyecto de múltiples componentes (frontend web, backend y aplicación móvil). Dado que el proyecto no contempló despliegue productivo durante el semestre, el análisis se enfoca en Integración Continua (CI) y generación/verificación de artefactos.

5.1 Criterios de evaluación

Considerando el alcance del proyecto, se definieron criterios observables y directamente verificables a través de los resultados de ejecución del pipeline:

1. Automatización de verificación por evento: ejecución del pipeline ante push y pull_request hacia ramas objetivo (principalmente main y Test).
2. Reproducibilidad: capacidad de construir y validar el proyecto en un entorno limpio (runner), independiente del entorno local de los desarrolladores.
3. Quality gates mínimos: verificación de instalación de dependencias, análisis estático/lint, ejecución de pruebas básicas y build/compilación.
4. Generación de artefactos verificables: construcción de outputs relevantes (por ejemplo, build del frontend, compilación del backend, construcción de imagen Docker y generación de APK debug).
5. Trazabilidad y *feedback*: disponibilidad de logs de ejecución y estados de aprobación/falla para apoyar la detección rápida de errores y la colaboración.

Estos criterios se alinean con el enfoque incremental descrito en el estado del arte: comenzar con validaciones mínimas y expandir cobertura a medida que el proyecto madura.

5.2 Resultados del pipeline por componente

5.2.1 Frontend web (React + TypeScript)

Para el componente web se implementó un workflow de CI con ejecución automática en push y pull_request sobre main y Test. El flujo parte con el checkout del repositorio y la configuración de Node.js en su versión 20, seguido de una instalación reproducible de dependencias mediante npm ci con uso de caché. A partir de ahí se ejecuta el lint cuando está disponible (que al inicio del proyecto se dejó deshabilitado, pero más tarde lo habilitamos), el build del proyecto y las pruebas, estas últimas con tolerancia ante la ausencia de scripts formales definidos.

Además, se incorporó un mecanismo de verificación básica del framework de pruebas (por ejemplo, comprobar la presencia y versión de Vitest), lo que resulta útil mientras el proyecto aún consolida su suite de tests.

Resultado principal: el pipeline permitió asegurar que, ante cada cambio integrado o propuesto, el frontend era construible y mantenía un estándar mínimo de validación, reduciendo el riesgo de errores que aparecen únicamente al compilar en una máquina distinta. El workflow corrió en total 46 veces, y demoró entre 21 y 42 segundos, con un promedio de 32 segundos por iteración.

5.2.2 Backend (Node.js + Express)

El backend cuenta con un workflow que se ejecuta igualmente en push y pull_request hacia main y Test. El flujo comienza con el checkout del repositorio y la configuración de Node.js (versión 18), seguido de la instalación de dependencias mediante npm install. Posteriormente se ejecutan el lint y las pruebas, ambos con tolerancia a la ausencia del script correspondiente cuando este no está definido.

Resultado principal: se estandarizó la verificación de que el proyecto puede instalar dependencias en un entorno limpio y ejecutar controles básicos cuando están disponibles. El uso de tolerancia (|| echo ...) permitió que el pipeline no bloqueara el avance cuando aún no se habían consolidado scripts de lint o tests en el repositorio, favoreciendo adopción temprana.

El workflow corrió en total 75 veces, y demoró entre 22 y 31 segundos, con un promedio de 27 segundos por iteración.

Observación técnica: a diferencia del frontend, el backend utiliza npm install en lugar de npm ci. Esto es funcional, pero reduce determinismo respecto a lockfiles; por tanto, se identifica como una mejora natural para fortalecer reproducibilidad.

5.2.3 Aplicación móvil (Flutter)

Para móvil se implementó un workflow con disparo en push y pull_request hacia main y Test, con un enfoque particular en validar el entorno Flutter y asegurar la compilación del proyecto. El flujo comienza instalando Flutter en su canal estable, versión 3.35.6, y

verificando dicha instalación mediante `flutter --version`, paso que permite detectar tempranamente problemas de configuración del runner antes de avanzar al resto del pipeline. A continuación, se obtienen las dependencias del proyecto (`flutter pub get`) y se ejecuta el análisis estático (`flutter analyze`), configurado en esta etapa para no bloquear el pipeline ante advertencias (`|| true`), de modo que el equipo pueda ir corrigiendo observaciones de forma progresiva sin frenar la integración. Posteriormente se ejecutan las pruebas (`flutter test --no-pub`), que incluyen un smoke test mínimo orientado a confirmar que el entorno de testing está correctamente configurado, y finalmente se compila un artefacto Android en modo debug (`flutter build apk --debug`) como verificación de que el proyecto construye de extremo a extremo.

El resultado principal de este workflow es la estandarización de un ciclo de verificación móvil que confirma que el proyecto instala sus dependencias correctamente, puede ejecutar pruebas automatizadas sin intervención manual, y compila un APK debug de forma reproducible en un runner ajeno al entorno local de cada desarrollador. Esto resulta especialmente valioso considerando que el build móvil suele ser uno de los puntos más sensibles a inconsistencias de entorno, diferencias de SDK, versiones de Flutter o configuraciones locales que funcionan en la máquina de un integrante, pero no en la de otro, por lo que contar con esta verificación automática reduce de forma directa un tipo de error históricamente difícil de diagnosticar a distancia dentro del equipo.

El workflow corrió en total 24 veces, y demora entre 6 min con 29 segundos y 12 minutos con 51 segundos, con un promedio de 8 minutos y 24 segundos por iteración.

1.1.1 Verificación de contenedor (Docker Image CI)

Se incorporó un workflow dedicado a validar la construcción de una imagen Docker mediante:

- Checkout del repositorio.
- Ejecución de docker build con etiquetado basado en timestamp.

Resultado principal: esta verificación permitió detectar tempranamente problemas relacionados con el Dockerfile y asegurar que el empaquetado del proyecto puede realizarse de manera consistente, sentando la base para una futura publicación a un registry y despliegue a ambientes.

Observación técnica: el esquema de etiquetado utilizado (timestamp únicamente) es funcional para verificar la construcción de la imagen, pero constituye una práctica subóptima de versionado: al no incorporar el commit asociado, dificulta trazar una imagen específica hasta el cambio de código que la originó. Existen esquemas más robustos, pero no están dentro del alcance del proyecto o esta tesina.

5.3 Discusión de resultados

De manera global, el pipeline implementado cumplió con los objetivos mínimos de CI para un proyecto multi-repositorio. En primer lugar, aportó feedback temprano: al ejecutarse en `pull_request` y `push`, entrega señales inmediatas sobre fallos de instalación, lint, tests o build antes de integrar cambios a las ramas objetivo. También ofreció reproducibilidad práctica, ya que la ejecución en runners limpios reduce el problema clásico de "funciona en mi máquina", y permitió una estandarización entre componentes: aunque cada uno requiere herramientas distintas (Node, Flutter, Docker),

se consolidó una lógica homogénea de instalar, analizar, testear y construir. Finalmente, habilitó una adopción incremental, dado que la tolerancia a la ausencia de scripts de lint o tests permitió introducir CI sin bloquear al equipo en una etapa donde la suite de pruebas todavía estaba en construcción.

Sin embargo, el pipeline también presenta limitaciones coherentes con el estado del proyecto: la cobertura de pruebas aún es baja, por lo que entrega principalmente confianza de construcción más que de correctitud funcional; no se ejecutan pruebas de integración con base de datos ni con servicios externos como MySQL o Firebase dentro del runner; y no existe una etapa de despliegue a staging o producción, por lo que el CD queda como preparación, limitado a artefactos y al build de Docker.

Los datos duros:

Antes del pipeline, verificar código nuevo requería desplegar el sistema completo en contenedores, proceso que tomaba al menos 25 minutos y la atención constante de un integrante. Con el pipeline, la verificación se dividió en workflows independientes por repositorio, cuyos tiempos promedio fueron de 32 segundos en el frontend, 27 segundos en el backend y 8 minutos y 24 segundos en la aplicación móvil.

Tabla 2. Ejecuciones de los workflows de CI en GitHub Actions

Repositorio / workflow	Ejecuciones	Exitosas	Fallidas	Tasa de fallas
Frontend (Frontend CI – Vite + Vitest)	46	38	8	17,4 %
Backend (Backend CI, Node.js CI y Docker Image CI)	75	53	22	29,3 %
Móvil (Flutter CI)	24	11	13	54,2 %
Total	145	102	43	29,7 %

Nota: la tasa de fallas corresponde al cociente entre ejecuciones fallidas y ejecuciones totales de cada repositorio.

Fuente: elaboración propia.

La tabla las ejecuciones de todos los workflow, con un total de 145 ejecuciones y una tasa de fallas del 29,7%. El componente móvil presenta la mayor tasa de fallas (54,2%) y esto es coherente con la mayor sensibilidad del Build de Flutter a la configuración del entorno.

5.4 Amenazas a la validez y limitaciones de la evaluación

Esta evaluación tiene varias limitaciones que hay que dejar claras. La primera es que, al no tener despliegue continuo, simplemente no podemos medir en producción cosas como deployment frequency o time to restore service, porque no hay un "en producción" del cual medirlas. También la cobertura de pruebas sigue siendo baja: los smoke tests ayudan a confirmar que el entorno está vivo, pero de ninguna manera reemplazan una suite real de pruebas unitarias, de integración o E2E, y eso es algo que quedó pendiente más por tiempo que por falta de intención. Otra limitación importante es que nunca emulamos las dependencias externas dentro de CI (ni base de datos ni servicios como Firebase), así que cualquier falla de integración real con esos servicios simplemente no la íbamos a detectar antes de que llegara a alguien probando manualmente. Además de esto, la medición de los 25min que tomaba levantar el proyecto entero a mano, es mas una estimación sumando todas sus partes que un tiempo cronometrado, a pesar de que la experiencia y testimonios del equipo indique que era ese el tiempo. Y por último está el tema de los "pasos no bloqueantes": usamos tolerancia en lint y analyze para no frenar al equipo mientras recién estábamos armando el pipeline, pero si ese comportamiento se queda así para siempre, en la práctica termina escondiendo problemas de calidad en vez de solo darnos tiempo para resolverlos; esa tolerancia del lint debe ser temporal, no permanente.

6 Evolución propuesta del pipeline CI/CD y prácticas DevOps según el crecimiento de FastPlan

Este capítulo propone una hoja de ruta para la evolución del pipeline CI/CD y las prácticas DevOps de FastPlan, alineándolas con las tres etapas de crecimiento definidas para el proyecto:

1. Salida a producción en la Región de Valparaíso
2. Expansión a otras regiones con un modelo de suscripción y cobros
3. Ampliación del producto hacia una versión para civiles e integración con otros servicios de emergencia.

El objetivo es que la automatización y operación del software escalen en madurez, seguridad y confiabilidad a medida que aumentan usuarios, datos, criticidad y complejidad del negocio.

6.1 Principios de escalamiento DevOps para FastPlan

Independiente de la etapa, la evolución propuesta se sostiene en cinco principios:

4. Ambientes definidos y separados: desarrollo, staging y producción, con configuración y secretos por ambiente.
5. Artefactos inmutables y trazables: cada versión desplegada debe corresponder a un artefacto generado por CI (contenedor/build) y asociado a un commit/tag.
6. Quality gates progresivos: comenzar con controles mínimos (lint, smoke, build) e incrementar a unitarias, integración, E2E, seguridad y performance.
7. Infraestructura como código (IaC): reproducibilidad del entorno cloud (redes, servicios, bases de datos, storage) y control de cambios.

8. Observabilidad y respuesta: monitoreo, logs, métricas y alertas que permitan mantener disponibilidad y recuperar ante fallos.

6.2 Etapa 1: salida a producción en Valparaíso

6.2.1 Recomendación de nube y arquitectura inicial

Para FastPlan, una opción coherente con el stack y el uso de Firebase es Google Cloud Platform (GCP), por su simplicidad operativa y compatibilidad. El backend Node/Express podría desplegarse en Cloud Run, un servicio serverless basado en contenedores que ofrece escalado automático y rollbacks sencillos, mientras que la base de datos MySQL encontraría en Cloud SQL for MySQL una alternativa administrada, con backups y alta disponibilidad opcional.

Para los planes en PDF y otros archivos, Cloud Storage permite organizar buckets por ambiente con políticas de acceso diferenciadas. En cuanto a autenticación, notificaciones y sincronización, se propone mantener Firebase (Auth, FCM, Firestore o Storage según el uso real) pero integrándolo de forma formal como parte de la arquitectura, en lugar de tratarlo como un componente adicional. Finalmente, el frontend web podría alojarse en Firebase Hosting o en Cloud Storage con CDN, ambas opciones rápidas, simples y de bajo costo.

6.2.2 Evolución del pipeline para producción

En esta etapa el pipeline debe evolucionar de "CI y artefactos" a un esquema de CI más CD controlado, incorporando ambientes diferenciados, aprobaciones y capacidad de recuperación ante fallos.

a) Ambientes

Se propone trabajar con tres ambientes claramente diferenciados: un ambiente de desarrollo (Dev) orientado a la integración frecuente de cambios, un ambiente de staging que replica de forma cercana las condiciones de producción y permite ejecutar pruebas de aceptación antes de liberar una versión, y el ambiente de producción propiamente tal, correspondiente al entorno real utilizado por bomberos en operación.

b) Workflows recomendados

El primer workflow actúa como quality gate en cada Pull Request, ejecutando lint, typecheck, pruebas unitarias y build, y bloqueando el merge en caso de que alguna de estas verificaciones falle; de esta forma, ningún cambio que rompa el proyecto llega a integrarse a la rama principal. Al hacer merge a main, se dispara un segundo workflow que construye la imagen Docker, la publica en un registry (ya sea GCP Artifact Registry o GHCR) y despliega automáticamente a staging, cerrando el ciclo con smoke tests post-deploy que verifican health checks y endpoints críticos antes de dar por buena la versión. Finalmente, la creación de un tag o release dispara el despliegue a producción, el cual en esta primera etapa contempla aprobación manual como medida de seguridad, junto con la capacidad de hacer rollback ante cualquier incidente.

c) Pruebas mínimas nuevas (prioridad alta)

Para sostener este esquema de despliegue con mayor confianza, se identifican como prioritarias las pruebas de endpoints críticos y de validación de contrato en el backend (por ejemplo, mediante OpenAPI si el equipo lo adopta), un conjunto mínimo de pruebas de componentes clave en el frontend además del build, y en el caso de móvil, mantener los smoke tests y el build existentes mientras se comienza a controlar la compatibilidad de API mediante versionado, de modo que cambios en el backend no rompan versiones de la app ya distribuidas.

d) Operación y confiabilidad

Sostener despliegues a producción exige, además, capacidad de observar y recuperar el sistema: esto implica logs centralizados en Cloud Logging junto con métricas y alertas sobre latencia, errores 5xx y disponibilidad, backups automáticos de la base de datos con sus respectivas políticas de retención y recuperación, y una gestión de secretos por ambiente a través de Secret Manager, evitando así que credenciales de staging y producción se mezclen o queden expuestas.

Resultado esperado de esta etapa: FastPlan se vuelve "operable", con despliegues repetibles, trazables y con capacidad de recuperación rápida ante fallos.

6.3 Etapa 2: Expansión Regional y Modelo con cobros

Esta etapa introduce dos cambios fuertes: crecimiento de usuarios y datos, y responsabilidad financiera por el manejo de cobros, lo que exige mayor madurez en seguridad, arquitectura y pipeline.

Entre las implicancias técnicas más relevantes están la necesidad de multi-tenancy para separar lógicamente los datos de distintas edificaciones, organizaciones o regiones con control de acceso estricto, la escalabilidad frente a un mayor tráfico y frecuencia de uso, y la integración de pagos a través de un proveedor especializado (PSP) como Stripe o MercadoPago, evitando manejar tarjetas directamente.

6.3.1 Evolución del pipeline (security y compliance by design)

A partir de aquí, el pipeline debe dar un salto de madurez e incorporar gates de seguridad y control de cambios más formales, ya que el manejo de pagos y datos sensibles deja de tolerar el nivel de informalidad aceptable en etapas anteriores.

En primer lugar, se incorpora SAST y análisis de dependencias, escaneando vulnerabilidades tanto en el stack de Node como en el de Flutter, con alertas automáticas y bloqueo del pipeline cuando se detecta una vulnerabilidad de severidad alta; a esto se suma el escaneo de las imágenes Docker, que revisa tanto la base image como las dependencias empaquetadas, dado que una imagen base desactualizada puede introducir riesgos independientemente de la calidad del código propio del proyecto. En paralelo, se formaliza el control de cambios mediante revisiones obligatorias vía CODEOWNERS, de modo que los PRs críticos (aquellos que tocan pagos, autenticación o permisos) requieran explícitamente la aprobación de los responsables de esas áreas antes de integrarse, y se adopta versionado semántico con releases formales, dejando

un historial de tags y changelog que permite rastrear qué cambió entre versiones y facilita el diagnóstico ante incidentes.

También se vuelve necesario controlar las migraciones de base de datos como un paso explícito del pipeline, aplicándolas primero en staging y, en producción, solo tras una validación previa y con un plan de rollback definido, evitando que un cambio de esquema mal calculado deje el sistema en un estado inconsistente. A nivel de pruebas, se incorporan pruebas de integración levantando una instancia efímera de MySQL como servicio de Docker dentro de CI, lo que permite validar endpoints que dependen realmente de la base de datos y no solo de mocks, y pruebas E2E selectivas concentradas en los flujos más críticos del sistema (login, suscripción, carga o actualización de un plan, y despacho), priorizando cobertura donde una falla tendría mayor impacto en el negocio o en la operación real de los bomberos.

6.3.2 Estrategia de despliegue

El staging se mantiene siempre actualizado desde main, funcionando como un reflejo constante del estado más reciente del proyecto. En producción, se adoptan estrategias de despliegue gradual (blue/green o canary) una vez que el volumen de usuarios lo amerite, reduciendo el riesgo de que un cambio problemático afecte a la totalidad de los usuarios de forma simultánea. A esto se suman feature flags, que permiten habilitar funciones específicas por región o cliente sin necesidad de un nuevo despliegue, dando flexibilidad para probar o liberar funcionalidades de forma controlada.

Resultado esperado de etapa 2: el pipeline no solo "compila", sino que protege el negocio frente a riesgos de pagos y seguridad, y habilita despliegues seguros a escala.

6.4 Etapa 3: Versión civil y expansión a múltiples servicios de emergencia

Esta etapa aumenta considerablemente la complejidad del sistema, tanto por el crecimiento de la base de usuarios hacia población civil como por la mayor heterogeneidad de los flujos que el sistema debe soportar. A esto se suma la posibilidad de integrar otras instituciones (ambulancias, policía) como consumidores o generadores de información dentro de la plataforma, y la exposición a picos de carga puntuales pero intensos, propios de emergencias masivas donde un gran número de usuarios interactúa con el sistema en un corto período de tiempo. Evolución de arquitectura y operación

6.4.1 Evolución de arquitectura y operación

Si el backend continúa creciendo, se vuelve razonable avanzar hacia una separación por dominios o servicios (por ejemplo, un servicio de planes, uno de despacho y otro de notificaciones), de modo que cada componente pueda escalar, desplegarse y fallar de forma independiente sin comprometer al resto del sistema. Esta separación, a su vez, hace indispensable una observabilidad más avanzada: trazas distribuidas que permitan seguir una solicitud a través de los distintos servicios, dashboards específicos por servicio, y SLOs que definan objetivos concretos de disponibilidad y latencia contra los cuales medir el desempeño real del sistema. Finalmente, la resiliencia pasa a ser un requisito explícito, incorporando colas o mecanismos basados en eventos para procesos como el

despacho o las notificaciones, junto con rate limiting y protección ante abuso, de forma que un pico de tráfico o un uso malintencionado no comprometa la disponibilidad del sistema para el resto de los usuarios.

6.4.2 Evolución del pipeline (madurez "producto")

El pipeline, por su parte, debe madurar en la misma dirección. Se incorporan pruebas de performance en staging bajo cargas representativas del uso real, y pruebas de seguridad más exigentes, que incluyen revisión de permisos y roles junto con hardening de las APIs expuestas. El pipeline móvil también evoluciona hacia un flujo de entrega más formal, con builds firmados para release, distribución interna a través de TestFlight o Play Internal Testing, automatización de la publicación cuando corresponda, y pruebas en dispositivos reales mediante un device farm para los flujos más críticos de la aplicación. Por último, se formaliza la gestión de incidentes mediante runbooks y postmortems, generando un ciclo de mejora iterativa guiado tanto por métricas DORA como por métricas operacionales propias del sistema.

Resultado esperado de etapa 3: FastPlan opera como plataforma robusta, con despliegues progresivos, control de calidad profundo, y capacidad real de sostener un producto masivo, brindando apoyo a la seguridad de servicios de emergencia y la ciudadanía.

6.5 Hoja de ruta priorizada (qué agregar primero)

Para que esta hoja de ruta sea implementable, una secuencia práctica sería la siguiente:

Etapas 1 Antes de la producción

Esta etapa se centra en dejar el sistema listo para operar de forma real, aunque sea a pequeña escala. Incluye la puesta en marcha de staging como ambiente estable, la contenerización de los componentes para asegurar consistencia entre entornos, y un flujo de CD que despliegue automáticamente a staging tras cada integración a main. A esto se suma el paso a producción propiamente tal, controlado mediante aprobación manual como medida de seguridad inicial, junto con logs y alertas que permitan detectar problemas operacionales a tiempo, y backups automáticos de la base de datos que garanticen la posibilidad de recuperar información ante un fallo.

Etapas 2 Antes de los cobros

Antes de manejar transacciones reales, el foco se traslada a la seguridad y la integridad de los datos. Esto implica incorporar SAST y dependency scanning para detectar vulnerabilidades de forma temprana, pruebas de integración contra una base de datos real que validen el comportamiento efectivo de los endpoints, y migraciones de base de datos controladas, aplicadas primero en staging y con plan de rollback antes de llegar a producción. Se suma también la gestión de secretos mediante un secrets manager dedicado, evitando exponer credenciales sensibles en el código o en configuraciones locales, y la adopción de versionado semántico, que deja trazabilidad clara de qué cambió entre una versión y otra del sistema.



Etapa 3 Antes de civiles

Finalmente, antes de abrir el sistema a una base de usuarios civiles más amplia, la prioridad es la capacidad de sostener carga y recuperarse de fallos sin afectar la operación. Esto incluye pruebas de performance que simulen condiciones de uso representativas, estrategias de despliegue gradual (como canary o blue/green) que permitan liberar cambios de forma progresiva y con bajo riesgo, y una observabilidad avanzada que dé visibilidad real del comportamiento del sistema en producción. Se complementa con un pipeline móvil "release-ready", capaz de generar builds firmados y distribuibles de forma confiable, y con la incorporación de gestión de incidentes, de modo que el equipo cuente con procesos claros para responder y aprender de fallas una vez que el sistema esté en operación real.

7 Conclusiones

Esta sección aborda, en primer lugar, los resultados técnicos del trabajo y, luego, los aspectos organizacionales y formativos asociados al proyecto.

La implementación del pipeline de Integración Continua tuvo un impacto positivo en el flujo de trabajo del equipo. Antes de su incorporación, verificar cualquier cambio requería levantar el sistema completo (web, backend, base de datos y móvil) en contenedores Docker, proceso que tomaba al menos 25 minutos y requería la atención constante de un integrante. Con el pipeline, cada repositorio pasó a validarse de forma independiente: el workflow del frontend tardó en promedio 32 segundos, el del backend 27 segundos y el de la aplicación móvil 8 minutos y 24 segundos. De esta forma, quien trabajaba en el backend ya no necesitaba montar el proyecto completo, incluida la aplicación móvil, para comprobar si sus cambios introducían errores.

Esto permitió que el equipo pasara de realizar pocos commits por sprint, dado que cada cambio se revisaba manualmente antes de subirlo, a integrar cambios con mucha mayor frecuencia: los tres workflows se ejecutaron 145 veces, con 102 ejecuciones exitosas y 43 fallidas (tasa de fallas de 29,7 %).

El objetivo general, proponer, adaptar e implementar un pipeline de Integración Continua con generación de artefactos para FastPlan, se cumplió, al igual que los objetivos específicos: se analizaron las necesidades del proyecto, se seleccionó y adaptó un modelo de pipeline utilizado en la industria, se implementaron workflows de CI en GitHub Actions para frontend, backend y móvil, se generaron artefactos reproducibles (build web, imagen Docker y APK debug) y se evaluó el impacto del pipeline mediante tiempos de validación y frecuencia de fallas. El despliegue continuo a staging o producción queda fuera del alcance y se plantea como la siguiente etapa de evolución (Capítulo 6).

Proyectando FastPlan más allá del contexto universitario, se concluye que una política de CI/CD y un pipeline robusto son clave para que un proyecto de TI pueda crecer, ya que su ausencia tiende a generar problemas de integración, calidad y coordinación que se agravan a medida que el proyecto escala.

En cuanto a la organización del equipo, las condiciones del primer sprint obligaron a trabajar jornadas extensas con plazos acotados, lo que impidió definir hábitos y una cultura de equipo desde el inicio. Cabe considerar que los integrantes cursaban otras asignaturas en paralelo, por lo que FastPlan no era su única prioridad, a diferencia de lo que ocurriría en un proyecto empresarial. Aun así, en los sprints siguientes se logró establecer un ritmo de trabajo común, y la incorporación del pipeline alivió varios aspectos del flujo de trabajo.

Finalmente, desde el punto de vista formativo, asumir los roles de Scrum Master y DevOps permitió al autor observar el proyecto desde una perspectiva global y aprender no solo sobre CI/CD y DevOps, sino también sobre las distintas aristas de un equipo de desarrollo y de un proyecto de software en su conjunto, experiencia cercana al ejercicio profesional de la Informática.

8 Referencias

- [1] Kim, G., Humble, J., Debois, P., Willis, J., The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations, Portland: IT Revolution Press, 2016.
- [2] M. Fowler, «Continuous Integration,» martinowler.com, 2010. [En línea]. Available: <https://martinfowler.com/articles/continuousIntegration.html>. [Último acceso: 2026].
- [3] Humble, J., Farley, D., Continuous Delivery: Reliable Software Releases through Build, Test, and Deployment Automation, Boston: Addison-Wesley, 2010.
- [4] «Trunk Based Development,» trunkbaseddevelopment.com, [En línea]. Available: <https://trunkbaseddevelopment.com/>.
- [5] V. Driessen, «A successful Git branching model,» nvie.com, ene. 2010. [En línea]. Available: <https://nvie.com/posts/a-successful-git-branching-model/>. [Último acceso: 2026].
- [6] ESLint, «Core Concepts,» ESLint Docs, [En línea]. Available: <https://eslint.org/docs/latest/use/core-concepts/>. [Último acceso: 2026].
- [7] Prettier, «What is Prettier?,» Prettier Docs, [En línea]. Available: <https://prettier.io/docs/>. [Último acceso: 2026].
- [8] TypeScript, «TypeScript for the New Programmer,» TypeScript Docs, [En línea]. Available: <https://www.typescriptlang.org/docs/handbook/typescript-from-scratch.html>. [Último acceso: 2026].
- [9] GitHub, Inc., «Workflow syntax for GitHub Actions,» GitHub Docs, [En línea]. Available: <https://docs.github.com/actions/using-workflows/workflow-syntax-for-github-actions>. [Último acceso: 2026].
- [10] GitHub, Inc., «Using secrets in GitHub Actions,» GitHub Docs, [En línea]. Available: <https://docs.github.com/actions/security-guides/using-secrets-in-github-actions>. [Último acceso: 2026].
- [11] Docker, Inc., «What is a container?,» Docker Docs, [En línea]. Available: <https://docs.docker.com/get-started/docker-concepts/the-basics/what-is-a-container/>. [Último acceso: 2026].
- [12] T. Preston-Werner, «Semantic Versioning 2.0.0,» semver.org, [En línea]. Available: <https://semver.org/>. [Último acceso: 2026].

- [13] Google, «Monitoring Distributed Systems,» Google SRE Workbook, [En línea]. Available: <https://sre.google/workbook/monitoring/>. [Último acceso: 2026].
- [14] GitHub, Inc, «Workflow syntax for GitHub Actions,» GitHub Docs, [En línea]. Available: <https://docs.github.com/actions/using-workflows/workflow-syntax-for-github-actions>. [Último acceso: 2026].
- [15] Microsoft, «Azure DevOps Pipelines baseline architecture,» Microsoft Learn, [En línea]. Available: <https://learn.microsoft.com/en-us/azure/devops/pipelines/architectures/devops-pipelines-baseline-architecture>. [Último acceso: 2026].
- [16] Alibaba Cloud, «GitHub Actions + ACK: A powerful combination for cloud-native DevOps implementation,» Alibaba Cloud Blog, [En línea]. Available: https://www.alibabacloud.com/blog/github-actions-%2B-ack-a-powerful-combination-for-cloud-native-devops-implementation_597623. [Último acceso: 2026].
- [17] SonarSource, «Introduction to quality gates,» Sonar Documentation, [En línea]. Available: <https://docs.sonarsource.com/sonarqube-server/latest/quality-standards-administration/managing-quality-gates/introduction-to-quality-gates/>. [Último acceso: 2026].
- [18] Codemagic, «Building a Flutter app,» Codemagic Docs, [En línea]. Available: <https://docs.codemagic.io/yaml-quick-start/building-a-flutter-app/>. [Último acceso: 2026].
- [19] DORA, «DORA's software delivery performance metrics,» dora.dev, [En línea]. Available: <https://dora.dev/guides/dora-metrics/>. [Último acceso: 2026].
- [20] Forsgren, N., Humble, J., Kim, G., Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations, Portland: IT Revolution Press, 2018.