

UNIVERSIDAD TÉCNICA FEDERICO SANTA MARÍA
DEPARTAMENTO DE ELECTRÓNICA E INFORMÁTICA
CONCEPCIÓN - CHILE



**“BeatMap: IMPLEMENTACIÓN DE SERVIDOR LOCAL,
CLOUD Y ARQUITECTURA DE SOFTWARE”**

DISMAR WILDER QUIROZ ZAVALETA

**MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO EN INFORMÁTICA**

Profesor Guía: Cristian Antonio Lara Valenzuela

Enero - 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;

Título del trabajo: BeatMap: Implementación de servidor local, cloud y arquitectura de software.

Nombre del candidato(a): Dismar Wilder Quiroz Zavaleta

Carrera / Grado: Ingeniería en informática

Campus: Concepcion ; **Departamento:** Electronica e Informatica

2.- VALIDACIÓN DEL PROFESOR GUÍA/DIRECTOR DE TESIS

Yo, Cristian Antonio Lara Valenzuela, 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

☒ 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:

☐ 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: 16/01/2025 ; **Firma:** 

Estudiante o Candidato(a):

Fecha: 16/01/2025 ; **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.

DEDICATORIA

Esta presente memoria se la quiero dedicar a mi familia, en especial a mis padres quienes en este proceso fueron las personas que con su apoyo incondicional siempre me alentaron a seguir adelante sin importar las adversidades que se pueda tener. A mi madre quien es la persona más importante en mi vida, la que siempre me apoyo en todo momento es por ti que en estos momentos estoy acá, la mujer que siempre creyó en mí, a mi padre que me enseñó los valores y principios que debo llevar en la vida, que a pesar de todo siempre me enseñó que el esfuerzo y el sacrificio tiene su recompensa a los dos les dedico esta memoria porque se lo merecen por ser buenos padres, y por fin puedo decir soy ingeniero gracias a ustedes.

AGRADECIMIENTOS

Primeramente agradecer a todos las personas que fueron parte de esto y que estuvieron del día uno y que siempre me apoyaron en este camino, a mis compañeros que se convirtieron en mis grandes amigos que siempre nos apoyábamos entre nosotros y no dejábamos a nadie atrás, a todos ellos que alguna vez creyeron en mí les agradezco de todo corazón, a mi familia que siempre me apoyaron y confiaron en mí que siempre me daban ánimos en esos días de estrés, en especial a mis padres que fueron y serán el pilar más importante en esta vida y que por ellos hoy soy lo que soy, a mi novia que en este proceso fue la que me supo comprender y la que supo escuchar en los momentos donde no podía más. A todos ustedes les agradezco por ser esto posible cada uno de ustedes fueron parte de este hermoso proceso muchas gracias.

RESUMEN

Resumen— El presente trabajo de título aborda la modernización del ciclo de vida de desarrollo de la aplicación "BeatMap", la cual sufría de procesos de despliegue manuales, lentos y propensos a errores. El objetivo principal fue diseñar e implementar una arquitectura de Integración y Despliegue Continuo (CI/CD) eficiente y de bajo costo. La metodología utilizada fue de implementación incremental por fases: 1) Estandarización de entornos mediante contenedores Docker; 2) Implementación de una infraestructura en la nube utilizando AWS EC2, externalización de base de datos con Supabase y seguridad perimetral con Cloudflare; y 3) Automatización del flujo de trabajo mediante pipelines en Jenkins. Los resultados principales demuestran una reducción del tiempo de despliegue de 60 minutos a menos de 5 minutos, la eliminación de errores humanos en la puesta en producción y la creación de un entorno de pruebas (QA) aislado. La relevancia de estos resultados radica en demostrar que es posible profesionalizar la operación de software en proyectos emergentes utilizando herramientas gratuitas y de código abierto, garantizando escalabilidad y seguridad sin incurrir en costos operativos elevados.

Palabras Clave— DevOps; CI/CD; AWS; Docker; Automatización.

ABSTRACT

Abstract—This thesis addresses the modernization of the development lifecycle of the "BeatMap" application, which suffered from manual, slow, and error-prone deployment processes. The main objective was to design and implement an efficient and low-cost Continuous Integration and Continuous Deployment (CI/CD) architecture. The methodology used was incremental implementation by phases: 1) Standardization of environments using Docker containers; 2) Implementation of cloud infrastructure using AWS EC2, database outsourcing with Supabase, and perimeter security with Cloudflare; and 3) Workflow automation using Jenkins pipelines. The main results demonstrate a reduction in deployment time from 60 minutes to less than 5 minutes, the elimination of human errors in production releases, and the creation of an isolated testing environment (QA). The relevance of these results lies in demonstrating that it is possible to professionalize software operations in emerging projects using free and open-source tools, ensuring scalability and security without incurring high operating costs.

Keywords— DevOps; CI/CD; AWS; Docker; Automation.

GLOSARIO

API (Application Programming Interface): Conjunto de definiciones y protocolos que se utiliza para desarrollar e integrar el software de las aplicaciones.

AWS (Amazon Web Services): Plataforma de servicios de computación en la nube ofrecida por Amazon.

BaaS (Backend as a Service): Modelo de servicio en la nube que permite a los desarrolladores externalizar la gestión de la base de datos y la infraestructura del lado del servidor (ej. Supabase).

CI/CD (Continuous Integration / Continuous Deployment): Práctica de desarrollo de software que consiste en automatizar la integración de cambios de código y su despliegue a producción.

DNS (Domain Name System): Sistema de nomenclatura jerárquico descentralizado para dispositivos conectados a redes IP, que traduce nombres de dominio a direcciones IP.

Docker: Plataforma de software que permite crear, probar e implementar aplicaciones rápidamente mediante el uso de contenedores.

EC2 (Elastic Compute Cloud): Servicio web que proporciona capacidad informática redimensionable en la nube de AWS.

HTTP/HTTPS (Hypertext Transfer Protocol Secure): Protocolo de comunicación de Internet que protege la integridad y la confidencialidad de los datos entre el ordenador del usuario y el sitio web.

IP (Internet Protocol): Etiqueta numérica asignada a cada dispositivo conectado a una red informática.

Jenkins: Servidor de automatización de código abierto que permite a los desarrolladores construir, probar y desplegar su software de forma fiable.

QA (Quality Assurance): Conjunto de actividades planificadas y sistemáticas aplicadas en un sistema de calidad para que los requisitos de calidad de un producto o servicio sean satisfechos.

SSH (Secure Shell): Protocolo de administración remota que permite a los usuarios controlar y modificar sus servidores remotos a través de Internet.

SSL/TLS (Secure Sockets Layer / Transport Layer Security): Protocolos criptográficos que proporcionan comunicaciones seguras por una red.

VPC (Virtual Private Cloud): Servicio que permite lanzar recursos de AWS en una red virtual definida por el usuario.

INDICE DE CONTENIDOS

RESUMEN	5
ABSTRACT.....	6
INDICE DE FIGURAS	11
INDICE DE TABLAS	12
INTRODUCCIÓN	13
CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA.....	14
CAPÍTULO 2: MARCO CONCEPTUAL.....	17
2.2 FUNDAMENTOS DEL ASEGURAMIENTO DE CALIDAD (QA)	18
2.2.1 Niveles y tipos de prueba	18
2.2.2 Entorno de QA aislado como requisito de DevOps	19
2.3 Análisis de Sistemas Operativos para Servidor de Pruebas (QA).....	20
2.4.1 EL PROBLEMA DE LA PORTABILIDAD DE ENTORNOS	22
2.4.2 TABLA COMPARATIVA Y JUSTIFICACION DE CONTENEDORES	23
2.4.4 DECISIÓN DE TECNOLOGÍA DE CONTENERIZACIÓN.....	23
2.5.1 MODELOS DE SERVICIO CLOUD: IAAS VS. PAAS.....	24
2.5.2 ALTERNATIVAS DE PROVEEDORES IAAS (NUBE PÚBLICA).....	25
2.5.4 DECISIÓN DE ARQUITECTURA Y PROVEEDOR CLOUD	26
2.6.3 DECISIÓN SOBRE HERRAMIENTAS CI/CD	28
2.7 ORQUESTACIÓN DE CONTENEDORES.....	29
2.7.1 ALTERNATIVAS CONSIDERADAS	29
2.7.2 TABLA COMPARATIVA Y JUSTIFICACIÓN DE ORQUESTADORES	30
CAPÍTULO 3: PROPUESTA DE SOLUCION	32
3.1 INTRODUCCIÓN Y METODOLOGÍA DE IMPLEMENTACIÓN.....	32
3.2 FASE 1: IMPLEMENTACIÓN DEL ENTORNO DE TESTING (QA).....	32
3.2.1 Configuración del Servidor Local	32
3.2.2 Contenerización de la Aplicación (Docker)	33
3.3 FASE 2: IMPLEMENTACIÓN DE LA ARQUITECTURA DE PRODUCCIÓN (AWS).....	36
3.3.1 Red (AWS VPC)	37
3.3.3 Cómputo e Instancia (EC2)	38
3.3.4 Gestión de DNS y Seguridad (Cloudflare)	39
3.4 FASE 3: AUTOMATIZACIÓN DEL PIPELINE CI/CD CON JENKINS	40

3.4.1 Configuración de Jenkins	40
3.4.2 Estrategia de Despliegue	41
3.4.3 Pipeline as Code (Jenkinsfile)	41
3.4.4 Resumen del Flujo DevOps	42
3.5 CONCLUSIÓN DE LA PROPUESTA.....	42
CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN	43
4.1 ESTRATEGIA DE VALIDACIÓN	43
4.2 VALIDACIÓN DEL PROYECTO DE FERIA (APLICACIÓN BEATMAP).....	43
4.3 VALIDACIÓN DEL SUB-PROYECTO (SOLUCIÓN DEVOPS)	43
4.3.1 Validación del Entorno de QA y Pipeline de Integración	44
4.3.2 Validación de Infraestructura de Producción (AWS + Cloudflare + Supabase) .	44
4.3.3 Validación del Despliegue en Producción (CI/CD)	45
4.4 ANÁLISIS COMPARATIVO DE RESULTADOS	45
4.4.1 Indicadores de Rendimiento	45
4.4.2 Análisis de Costos y Recursos	46
4.5 CONCLUSIONES DE LA VALIDACIÓN	46
CAPÍTULO 5: CONCLUSIONES	47
5.1 CONCLUSIONES GENERALES	47
5.2 VALIDACIÓN DE LOS OBJETIVOS	48
5.3 RESULTADOS E IMPACTO DE LA SOLUCIÓN	48
5.4 ALCANCES Y LIMITACIONES	49
5.5 CONTRIBUCIONES Y APLICACIONES DEL TRABAJO.....	49
5.6 RECOMENDACIONES Y TRABAJOS FUTUROS	50
REFERENCIAS BIBLIOGRÁFICAS	51
ANEXOS	ERROR! BOOKMARK NOT DEFINED.

INDICE DE FIGURAS

Figura 1:Ciclo de vida de DevOps.	16
Figura 2:Niveles y Tipos de prueba QA	18
Figura 3: Sistema Operativos Server.	19
Figura 4: Flujo típico de Jenkins.	27
Figura 5: Dockerfile de BeatMap.	32
Figura 6: Docker-compose.yml de BeatMap.	33
Figura 7: Infraestructura DevOps/QA.	34
Figura 8: Infraestructura BeatMap Producción.	35
Figura 9: Arquitectura de Red (VPC).	36
Figura 10: Registro DNS de Cloudflare.	38
Figura 11: Configuración inicial de Jenkins.	39
Figura 12: Pipeline inicial de Jenkins.	40

INDICE DE TABLAS

Tabla 1: Tabla de objetivos específicos	15
Tabla 2: Tabla de comparación de SO.	20
Tabla 3: Tabla de comparación contenedores.	22
Tabla 4: Tabla de comparación arquitectura cloud.	24
Tabla 5: Tabla de comparación integración continua.	26
Tabla 6: Tabla de comparación Orquestadores.	29
Tabla 7: Tabla de definición comandos Docker.	32
Tabla 8: Comparación de tiempo y flexibilidad del despliegue.	44
Tabla 9: Análisis de recursos de infraestructura.	45

INTRODUCCIÓN

En el contexto actual de la ingeniería de software, la velocidad y la fiabilidad con la que se entregan nuevas funcionalidades son factores determinantes para el éxito de cualquier aplicación. La industria ha evolucionado desde modelos de desarrollo rígidos y monolíticos hacia metodologías ágiles que demandan ciclos de entrega continuos. Sin embargo, esta agilidad en el desarrollo a menudo choca con cuellos de botella en la etapa de operaciones, donde los despliegues manuales, la falta de estandarización entre entornos y la ausencia de pruebas automatizadas generan riesgos significativos de estabilidad y seguridad.

El presente trabajo de título se enmarca en el área de **DevOps (Development and Operations)**, una disciplina que busca unificar el desarrollo de software (Dev) y la operación del mismo (Ops). Específicamente, este proyecto aborda la modernización de la infraestructura y el ciclo de despliegue de la aplicación "BeatMap".

La problemática central que motiva esta memoria radica en la naturaleza "artesanal" y poco escalable del proceso de despliegue original de BeatMap. La dependencia de configuraciones manuales, la inexistencia de un entorno de pruebas (QA) aislado y el alojamiento precario de la aplicación no solo consumían tiempo valioso del equipo de desarrollo, sino que también exponían el servicio a caídas y errores en producción difíciles de diagnosticar. El alcance de este problema limitaba la capacidad del proyecto para crecer y profesionalizarse.

Como propuesta de solución, se plantea el diseño e implementación de una **Arquitectura de Integración y Despliegue Continuo (CI/CD)**. Esta solución no se limita a la adopción de herramientas, sino que propone una reingeniería del flujo de trabajo. Se busca transicionar desde un despliegue manual hacia una infraestructura automatizada basada en la nube, utilizando tecnologías de contenerización como Docker, orquestación de flujos con Jenkins y servicios cloud eficientes como AWS y Supabase.

La metodología empleada para el desarrollo de esta memoria fue de carácter **incremental y por fases**. En lugar de intentar una migración masiva y arriesgada, se procedió a construir la solución capa por capa: primero, estabilizando el entorno local mediante contenedores; segundo, diseñando una infraestructura productiva costo-eficiente en la nube; y finalmente, conectando ambos mundos mediante tuberías de automatización (pipelines). Este enfoque permitió validar cada componente técnico por separado antes de su integración final.

CAPÍTULO 1: DEFINICIÓN DEL PROBLEMA

1.1 Contexto General

Esta memoria se desarrolla bajo el marco del proyecto de BeatMap, una plataforma digital enfocada en el networking de músicos con productores, además con la necesidad de entregar un espacio de visibilización para artistas emergentes y bandas independientes con un enfoque en la difusión de eventos locales con la geolocalización de sus presentaciones para conectar con sus audiencias.

En particular, este subproyecto se sitúa en el área de la **Ingeniería del Software orientada a operaciones y despliegue (DevOps)**, abarcando la infraestructura técnica y los procesos que permiten llevar el sistema desde un entorno de desarrollo hasta su uso en producción. Actualmente, BeatMap no contaba con una estructura formal para el despliegue de sus servicios, lo que representa una limitación crítica para su evolución y escalabilidad.

1.2 Situación Actual y Problemática

Hasta antes de esta memoria, los entornos de desarrollo y pruebas del sistema se gestionaban de manera manual y local, sin ambientes diferenciados para Testing, QA o Producción. Esto generaba ineficiencias operativas, mayores tiempos de despliegue, y riesgos elevados de errores al mover versiones a producción.

Los principales actores involucrados en esta problemática son los integrantes del equipo de desarrollo de BeatMap, los encargados del aseguramiento de calidad (QA) y, finalmente, los usuarios finales que interactúan con la plataforma en su versión mas reciente. La ausencia de una arquitectura estable y un pipeline automatizado compromete tanto la experiencia del usuario como la eficiencia de nuestro equipo técnico.

1.3 Justificación

Este problema no es exclusivo de BeatMap. En múltiples proyectos de software en fase de crecimiento, la falta de procesos automatizados de despliegue e infraestructura escalable es una barrera común que retrasa la consolidación de los productos. Según un informe de

“Puppet State of DevOps (2023), las organizaciones con pipelines de CI/CD definidos despliegan hasta 208 veces más rápido que aquellas con procesos manuales y locales, lo que evidencia la necesidad de adoptar prácticas de integración y entrega continua desde etapas tempranas del ciclo de vida del software (p 22, 23).”

Si este problema no es abordado en el corto plazo, se corre el riesgo de que BeatMap enfrente dificultades críticas como tiempos prolongados para liberar nuevas funcionalidades, fallas imprevistas en producción y escasa capacidad de reacción ante incidentes. Esto afectaría tanto su desarrollo interno del proyecto y como a la funcionalidad frente a los usuarios, dificultando su proyección y escalamiento.

1.4 Objetivo General

El objetivo de esta memoria es **implementar una arquitectura integral para el despliegue de BeatMap**, que contemple entornos diferenciados para Testing y Producción, con procesos automatizados mediante Jenkins.

1.5 Objetivos Específicos

1. Implementar y configurar un entorno de servidor local orientado a pruebas de Testing y Aseguramiento de Calidad (QA).
2. Diseñar e implementar una arquitectura escalable y robusta del proyecto para su despliegue en un entorno cloud.
3. Configurar e integrar un pipeline de despliegue automatizado utilizando Jenkins como herramienta de integración continua.

Este subproyecto busca sentar las bases para una operación sostenible y profesional del sistema, permitiendo reducir tiempos de entrega, minimizar errores en producción y mejorar la experiencia del usuario final mediante procesos técnicos sólidos y reproducibles.

1.5.1 Tabla de objetivos

Objetivos Específico	Marco Conceptual	Propuesta de Solución
Implementar y configurar un entorno de servidor local orientado a pruebas de Testing y Aseguramiento de Calidad (QA).	- Entornos de desarrollo y testing- Infraestructura local. - Despliegue de contenedores. - Aseguramiento de Calidad (QA).	- Instalación y configuración de servidor local Ubuntu server con Docker y herramientas de testing. - Creación de entorno aislado para pruebas funcionales y de integración.
Diseñar e implementar una arquitectura escalable y robusta del proyecto para su despliegue en un entorno cloud.	- Arquitecturas cloud (IaaS, PaaS). - Escalabilidad y alta disponibilidad. - Balanceo de carga. - Seguridad en la nube.	- Selección de proveedor cloud. - Diseño de arquitectura distribuida. - Despliegue en instancias virtuales y configuración de recursos cloud. - Configuración de DNS y acceso seguro(SSL).
Configurar e integrar un pipeline de despliegue automatizado utilizando herramientas de integración y entrega continua como herramienta de automatización.	- Integración continua (CI). - Entrega continua (CD). - Herramientas de integración y entrega continua: pipelines, agentes y plugins. - DevOps y automatización.	- Diseño e implementación de Jenkinsfile. - Integración con GitHub para triggers automáticos. - Despliegue automatizado en QA y producción.

Tabla 1: Tabla de objetivos específicos.
Fuente: Elaboración Propia.

CAPÍTULO 2: MARCO CONCEPTUAL

En este capítulo se presentara los conceptos fundamentales que sustentan el desarrollo del subproyecto “Implementación de servidor local, cloud y arquitectura de software” de BeatMap. Se abordan temas tales como infraestructura local, contenedores, gestion de calidad, arquitectura de servidor en cloud, automatización e integración continua(CI/CD).

2.1 DevOps Concepto

DevOps es un enfoque sociotécnico que integra desarrollo (Dev) y operaciones (Ops) para acortar el ciclo de entrega y elevar la calidad y la confiabilidad del software mediante colaboración, automatización y medición continua [Humble10]. Más que un rol aislado, es un modelo de trabajo que combina prácticas (CI/CD, IaC, observabilidad, pruebas automatizadas), cultura (equipos multifuncionales, ownership compartido) y herramientas (pipelines, contenedores, nubes públicas) [Fowler20].

2.1.1 Ciclo de vida de DevOps

Debido al carácter continuo de DevOps, se usa un bucle infinito para representar que las fases del ciclo de vida de DevOps se relacionan entre sí. A pesar de que parece fluir de forma secuencial, este bucle simboliza la necesidad de colaboración constante y mejora iterativa a lo largo de todo el ciclo de vida.

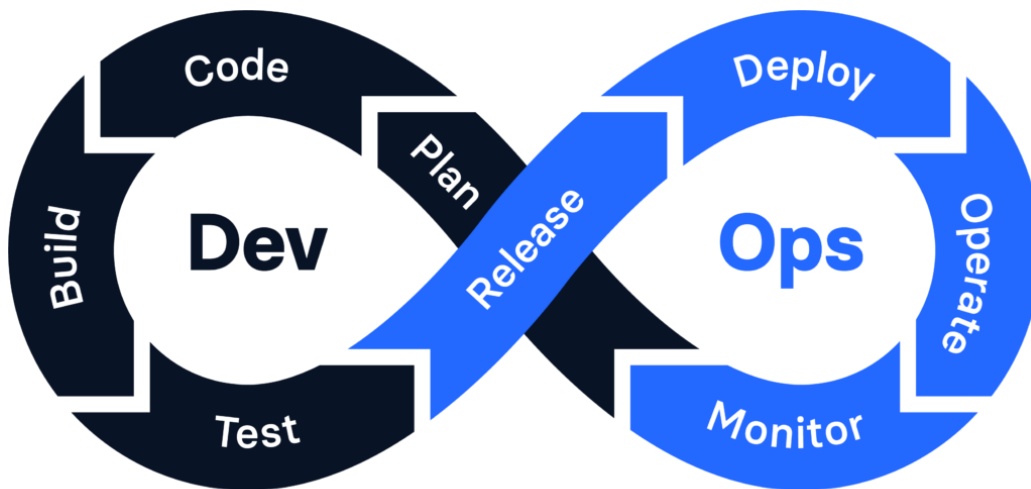


Figura 2:Ciclo de vida de DevOps.

Fuente: Atlassian.

2.2 FUNDAMENTOS DEL ASEGURAMIENTO DE CALIDAD (QA)

El **Aseguramiento de Calidad (Quality Assurance, QA)** en ingeniería de software es el conjunto de actividades planificadas y sistemáticas orientadas a **prevenir defectos** y a **proveer evidencia objetiva** de que el producto cumpla los requisitos funcionales y no funcionales a lo largo del **ciclo de vida del software (SDLC)**. A diferencia del **control de calidad** —enfocado principalmente en la detección de fallos al final—, QA pone el acento en **procesos, estándares y prácticas preventivas** (revisiones, automatización, criterios de aceptación, trazabilidad) para reducir el coste de corrección y mejorar la confiabilidad global del sistema [Pressman21].

En el contexto DevOps, QA opera como un **gate** del flujo CI/CD: cada cambio pasa por **verificación automatizada** en un entorno aislado antes de cualquier despliegue a producción. Este enfoque **shift-left** traslada pruebas y revisiones hacia etapas tempranas del **SDLC**, disminuyendo el tiempo de ciclo y la probabilidad de incidentes en operación. Los pilares que sostienen QA en DevOps son:

- **Definición como código** (pipelines).
- **Entornos reproducibles** (infraestructura y datos de prueba gestionados como código).
- **Métricas y observabilidad** (cobertura, tasa de fallos por despliegue, lead time, MTTR).
- **Trazabilidad** (matriz requisito → caso de prueba → evidencia), indispensable para auditoría y mejora continua.

2.2.1 Niveles y tipos de prueba

- **Pruebas funcionales:** verifican el cumplimiento del comportamiento especificado (casos de uso, reglas de negocio). Incluyen **unitarias** (a nivel de módulo), **integración** (interacción entre componentes) y **sistema** (fin a fin).
- **Pruebas de aceptación:** validan criterios del usuario/cliente en escenarios representativos.
- **Pruebas no funcionales:** evalúan atributos de calidad como **rendimiento** (latencia, throughput), **confiabilidad/disponibilidad**, **seguridad** (autenticación, autorización, OWASP), **usabilidad** y **compatibilidad**.

2.2.2 Entorno de QA aislado como requisito de DevOps

Un **entorno de QA aislado** evita efectos colaterales entre desarrollos simultáneos y producción, garantiza **reproducibilidad** de fallos y permite **datos de prueba controlados** (anonimizados o sintéticos). En DevOps, este entorno debe:

1. Ser **idéntico** (o muy similar) a producción en versiones de SO, runtime, librerías y configuración.
2. Permitir **provisionamiento automático** (infraestructura como código) y **semillas de datos** para pruebas repetibles.
3. Integrarse al pipeline como etapa **obligatoria** previa a release.
4. Exponer **telemetría** (métricas y logs) para analizar degradaciones antes de promover artefactos.

Este marco justifica, en los apartados siguientes, la selección y comparación de **sistemas operativos de servidor, contenedores y plataformas de automatización**.

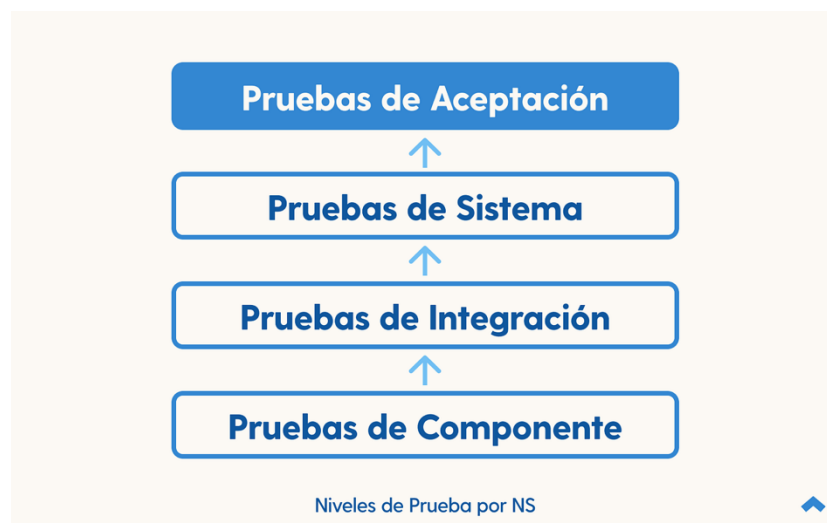


Figura 2:Niveles y Tipos de prueba QA.
Fuente: Medium.

2.3 Análisis de Sistemas Operativos para Servidor de Pruebas (QA)

Lo primero que se estableciera en un **servidor local/on-premise** es su propósito, que nos ofrece un entorno **controlado, de bajo costo y acceso inmediato** para validar *builds* antes de consumir recursos *cloud*. Requisitos mínimos:

- **Aislamiento:** redes, usuarios y secretos separados de producción; datos de prueba anonimizados.
- **Compatibilidad:** versiones de *runtime* (JDK/Node/Python), base de datos y *reverse proxy* equivalentes a producción.
- **Observabilidad:** logs centralizados y métricas básicas para analizar impactos de cada *build*.

Este entorno acelera el **ciclo de feedback** y reduce el costo de fallas al detectar defectos antes de los despliegues *cloud*.

Estos son los Sistemas Operativos a analizar y comparar en la siguiente seccion:



Figura 3: Sistema Operativos Server.
Fuente: Google Imagenes.

2.3.1 Alternativas de Sistemas Operativos de Servidor

A continuación se presenta un **análisis comparativo** de las opciones más comunes para servidores de QA, con enfoque en compatibilidad con **contenedores** y **Deploy**.

Criterio	Ubuntu Server (Linux)	Windows Server
Descripción general	Distribución Linux open source, ampliamente utilizada en entornos DevOps, servidores cloud y despliegues con contenedores. Ideal para pipelines CI/CD.	Sistema operativo propietario de Microsoft, enfocado en entornos corporativos y soluciones .NET. Integración nativa con Active Directory, IIS, y ecosistema Azure.
Licenciamiento y costo	Gratuito, con soporte opcional de Canonical (Ubuntu Pro). Ideal para entornos de desarrollo, pruebas y producción sin costos adicionales.	Requiere licencia comercial. Costos adicionales por usuario o instancia. Adecuado para empresas con infraestructura Microsoft existente.
Soporte para contenedores y orquestadores	Soporte nativo y estable para Docker , Podman , y Kubernetes (kubeadm, microk8s, k3s) . Es la base preferida para clusters cloud-native. [Ubuntu]	Soporte de contenedores limitado a Windows Containers y WSL2 . Kubernetes en Windows requiere nodos Linux para componentes principales. [Windows Server]
Rendimiento en entornos CI/CD	Ligero, rápido en tareas de compilación, despliegue y ejecución de contenedores. Menor consumo de recursos.	Más pesado en procesos de build y despliegue. Mayor consumo de memoria y CPU en servicios CI/CD concurrentes.
Ventajas principales	- Ligero y gratuito - Altamente personalizable - Ecosistema DevOps maduro - Soporte nativo para Docker/Kubernetes - Ideal para automatización e infraestructura cloud-native	- Integración nativa con Azure y Active Directory - Excelente administración gráfica - Soporte empresarial y documentación oficial - Ideal para entornos .NET o Windows legacy
Desventajas principales	- Curva de aprendizaje para quienes provienen del entorno Windows - Menor soporte para aplicaciones .NET o específicas de Microsoft	- Costos de licenciamiento - Menor rendimiento en CI/CD intensivo - Limitado soporte nativo para contenedores Linux - Ecosistema menos flexible

Tabla 2: Tabla de comparación de SO.

Fuente: Elaboración Propia.

2.3.2 Decisión de Sistema Operativo

Tras analizar las alternativas, **se seleccionó Ubuntu Server (LTS)** para el entorno de **QA local**. Esta decisión se fundamenta en:

- **Costo de licenciamiento nulo**, lo que favorece la sostenibilidad del entorno de pruebas y habilita la creación de réplicas/VMs adicionales sin impacto presupuestario.
- **Amplia documentación y comunidad activa** que reduce la curva de resolución de incidencias y acelera la puesta en marcha de **toolchains** modernos [Canonical23].
- **Compatibilidad nativa y madura con contenedores** (Docker/Podman) y su ecosistema — pilar central en esta memoria para lograr entornos reproducibles y despliegues consistentes (véase 2.3)—.

En síntesis, **Ubuntu Server (LTS)** ofrece el mejor equilibrio entre **coste, rapidez de adopción, compatibilidad con contenedores y alineación con prácticas DevOps**, por lo que se adopta como **SO de referencia para el servidor de QA local**, dejando documentados los criterios y las mitigaciones asociadas.

2.4 Análisis de Tecnologías de Contenerización

La contenerización surge como respuesta a la necesidad de **portabilidad, reproducibilidad y aislamiento** de aplicaciones a través de distintos ambientes (desarrollo, QA y producción). Su adopción dentro de prácticas **DevOps** posibilita pipelines CI/CD más confiables y despliegues consistentes al empaquetar **aplicación + dependencias** en unidades ejecutables estandarizadas [Merkel14].

2.4.1 El Problema de la Portabilidad de Entornos

En proyectos de software es recurrente ver el fenómeno conocido como “**Dependency Hell**” o *infierno de dependencias*: la aplicación funciona en el equipo del desarrollador pero **falla en QA o producción** debido a divergencias en versiones de librerías, configuración del sistema, *runtimes*, variables de entorno o binarios del sistema (“**en mi máquina sí funciona**”).

Estas discrepancias aumentan el **tiempo de ciclo**, introducen **defectos difíciles de reproducir** y encarecen el proceso de liberación. La **virtualización ligera** basada en contenedores mitiga el problema al proporcionar **entornos autocontenidos**, con **capas de imagen** versionadas y un **manifiesto** explícito de dependencias, lo que permite ejecutar el mismo artefacto en cualquier host compatible con el *runtime* de contenedores [Merkel14].

2.4.2 Tabla comparativa y Justificación de contenedores

Característica	Docker	Podman
Arquitectura	Cliente–Servidor con Daemon (Docker Engine).	Daemonless (sin demonio central); soporte rootless.
Ecosistema	Estándar de industria; Docker Hub con enorme catálogo.	Ecosistema creciente ; compatible con imágenes OCI .
Integración (tools)	Integración muy amplia con Jenkins, K8s, GH Actions, GitLab CI, etc.	Alta, especialmente en entornos Red Hat; a veces requiere más configuración.
Curva de aprendizaje	Baja (documentación/ejemplos abundantes).	Baja (CLI compatible/alias de Docker).
Seguridad	Requiere configurar rootless/endurecimiento adicional.	Más seguro por diseño (sin daemon; rootless nativo).

Tabla 3: Tabla de comparación contenedores.
Fuente: Elaboración Propia.

Docker prioriza **madurez del ecosistema** y **experiencia de uso** con integraciones listas; Podman privilegia **seguridad** y ejecución **sin Daemon**, manteniendo compatibilidad OCI.

2.4.4 Decisión de Tecnología de Contenerización.

A pesar del avance de alternativas como **Podman**, se **opta por Docker** como tecnología principal de contenerización debido a:

1. **Madurez y amplitud del ecosistema**, con un **catálogo masivo** de imágenes preconstruidas en **Docker Hub**, lo que acelera *builds* y prototipado.
2. **Integración nativa y probada** con la **gran mayoría de herramientas DevOps** (entre ellas **Jenkins, GitHub Actions, GitLab CI** y **Kubernetes**), simplificando la construcción de **pipelines CI/CD** y el paso a **orquestradores**.
3. **Curva de aprendizaje baja** y documentación extremadamente amplia, útil para equipos en crecimiento y para reducir tiempos de adopción.

En síntesis, **Docker** ofrece el mejor balance entre **portabilidad, ecosistema, integración con CI/CD y rapidez operativa** para los objetivos de este subproyecto. La evaluación de **Podman** se mantiene como línea futura —especialmente en entornos que prioricen **rootless** y **endurecimiento**—, sin afectar la decisión de corto plazo basada en **madurez y productividad** [Merkel14].

2.5 Análisis de Arquitectura y Proveedores Cloud

Esta sección desarrolla los **modelos de servicio cloud** relevantes para el subproyecto y compara **proveedores IaaS de nube pública** con criterios orientados a una arquitectura **DevOps** con CI/CD, despliegues reproducibles y escalabilidad.

2.5.1 Modelos de Servicio Cloud: IaaS vs. PaaS

IaaS (Infrastructure as a Service). El proveedor entrega infraestructura base: cómputo (VMs), red (VPC, balanceadores), almacenamiento y mecanismos de seguridad. El usuario gestiona el sistema operativo, runtime y aplicación. Ejemplos: AWS EC2 y VPC, Azure Virtual Machines, GCP Compute Engine.

PaaS (Platform as a Service). El proveedor gestiona **SO**, runtime, parches y escalado básico; el usuario se concentra en desplegar el código con menor fricción operativa. Ejemplos: Heroku, AWS Elastic Beanstalk / App Runner, GCP App Engine, Azure App Service.

Aunque PaaS simplifica la operación inicial, IaaS otorga control granular sobre:

1. Red y seguridad (subredes públicas/privadas, SG/NACL, bastiones, WAF),
2. Balanceo y ruteo (L4/L7, health checks, sticky sessions, blue/green y canary),
3. Sistema operativo y runtime (versionado exacto, hardening, agentización de observabilidad),
4. Topologías de disponibilidad (Multi-AZ, autoscaling).

En una memoria cuyo objetivo es diseñar e implementar una arquitectura DevOps personalizada y optimizar escalabilidad horizontal, IaaS es preferible: permite alinear PROD, instrumentar pipelines CI/CD con mayor control de artifacts, definir políticas de seguridad, y ajustar el costo-rendimiento de cada capa [Buyya19]. PaaS puede considerarse para microservicios específicos o picos de demanda cuando convenga priorizar time-to-market sobre control.

2.5.2 Alternativas de Proveedores IaaS (Nube Pública)

Amazon Web Services (AWS). **Lidera el mercado con un portafolio muy amplio y maduro. Tiene Componentes clave para esta memoria: EC2 (VMs), VPC (red), ALB/ELB (balanceo), RDS/Aurora (bases de datos gestionadas), S3 (almacenamiento), Route 53 (DNS), CloudWatch/CloudTrail (observabilidad/auditoría). Gran ecosistema, documentación y casos de uso.**

Microsoft Azure. Tiene una **Integración** profunda con el ecosistema Microsoft (**AD, Windows Server, .NET**), y **catálogo IaaS/PaaS competitivo**: Virtual Machines, Load Balancer/Application Gateway, Azure SQL, Blob Storage, Azure Monitor/Log Analytics. **Ventaja en identidad corporativa y compliance cuando el cliente opera con Microsoft.**

Google Cloud Platform (GCP). **Fortalezas en Big Data/IA y contenedores (pioneros con Kubernetes; GKE es referencia de mercado). IaaS con Compute Engine, Cloud Storage, Cloud SQL; observabilidad con Cloud Logging/Monitoring. Buenas prácticas de red y precio por segundo en cómputo.**

2.5.3 Tabla Comparativa y Justificación

Criterio	AWS	Azure	GCP
Cuota de mercado	Más alta (líder)	Segunda	Tercera (crecimiento sostenido)
Complejidad percibida	Alta (catálogo extenso)	Media-Alta	Media
Documentación y comunidad	Extensa, muchos <i>blueprints</i> y <i>workshops</i>	Muy buena, sobre todo para entornos MS	Muy buena, énfasis en K8s/Data
Fuerzas distintivas	Amplitud de servicios, ecosistema, madurez operacional	Identidad MS, integración AD/.NET, compliance corporativo	Kubernetes/GKE, analítica/IA, redes eficientes
Riesgos típicos	Sobrecarga de opciones si no se restringe catálogo	Costeo/licencias en <i>stacks</i> mixtos	Portafolio más focalizado; algunas integraciones empresariales menos maduras

Tabla 4: Tabla de comparación arquitectura cloud.

Fuente: Elaboración Propia.

2.5.4 Decisión de Arquitectura y Proveedor Cloud

Para la arquitectura de producción de BeatMap se selecciona **AWS** como proveedor **IaaS**. La decisión se fundamenta en:

1. La madurez de servicios **core** (EC2, VPC, **ALB/ELB** para **balanceo L4/L7**, **RDS/Aurora** para base de datos gestionada, **S3** y **Route 53**), que habilitan una arquitectura robusta y escalable.
2. Ecosistema y documentación muy amplios, con patrones probados para alta disponibilidad (**Multi-AZ**), **autoscaling**, observabilidad (**CloudWatch**, **CloudTrail**) y seguridad (IAM, KMS, Security Groups, NACL).
3. Alineación con DevOps: integración fluida con **pipelines CI/CD**, **registros de contenedores** (ECR), y opciones para evolucionar a **CaaS** (EKS) sin reescrituras profundas del *stack*.

Mitigaciones:

- **Costos** por consumo inadvertido. *Mitigación:* **presupuestos/alerts**, **Savings Plans/RI** cuando aplique, políticas de **apagado automático** en no-producción.

Conclusión. En virtud de los objetivos del subproyecto (alta disponibilidad, eficiencia y escalabilidad con **control granular** de red/seguridad y **flujo DevOps** personalizado), **AWS (IaaS)** provee el mejor **balance entre control, madurez y ecosistema** para la **plataforma de producción** de BeatMap, dejando abiertas líneas de evolución (EKS/GitOps, CDN/WAF, caché administrada) que se desarrollarán en el Capítulo 3.

2.6 Análisis de Herramientas de Integración Continua (CI/CD)

La **Integración Continua (CI)** es la práctica DevOps de integrar cambios de código con frecuencia en un repositorio compartido, verificándolos mediante **pruebas automatizadas** para detectar defectos de forma temprana, reducir el **lead time** y mejorar la calidad de las entregas [Humble10]. La **Entrega/Despliegue Continuo (CD)** extiende la automatización hacia la **promoción y publicación** de artefactos en entornos superiores (QA/Producción), con *gates* de calidad y aprobaciones controladas.

Un **pipeline** es la **representación como flujo** (generalmente declarativo) de las etapas de **build → test → package → scan → deploy**, incluyendo condiciones, aprobaciones y *rollbacks*. En el enfoque “**pipelines as code**”, su definición se **versiona junto al código** de la aplicación, habilitando *code review*, trazabilidad y reproducibilidad del proceso [Fowler20].

2.6.1 Alternativas de Herramientas CI/CD

La **Integración Continua (CI)** constituye una práctica esencial dentro del enfoque DevOps, la cual facilita la integración frecuente de modificaciones en el código o nuevas funcionalidades en un repositorio compartido. Este proceso se verifica mediante la ejecución de pruebas automatizadas, lo que contribuye a la reducción significativa de errores, la detección temprana de fallos y la aceleración del ciclo de desarrollo.

Jenkins. **Estándar on-premise histórico (código abierto, Java)** con ecosistema masivo de plugins. **Destaca por su flexibilidad y capacidad de integrar flujos complejos multi-repos, multi-agente y multi-entorno (self-hosted).** Ideal cuando se requiere control total del ambiente de ejecución, conectividad privada y personalización fina del pipeline.

GitLab CI/CD. **Solución** integrada al repositorio **GitLab**; define pipelines en **.gitlab-ci.yml (YAML)**. Ofrece runners compartidos (SaaS) o propios (self-hosted), integración nativa con issues, MR y *environments*. Potente en organizaciones que ya estandarizan en GitLab.

GitHub Actions. **CI/CD** embebido en GitHub, con *workflows* **YAML** y un marketplace amplio de *actions*. Excelente time-to-value cuando el código ya está en GitHub; runners hospedados reducen mantenimiento, con opción de *self-hosted runners*.

2.6.2 Tabla Comparativa y Justificación

Herramienta	Jenkins	GitLab CI	GitHub Actions
Hosting	Self-Hosted (requiere servidor)	Integrado (SaaS) o Self-Hosted	Integrado (SaaS); runners propios opcionales
Flexibilidad	Extremadamente alta (plugins)	Alta (integrada a GitLab)	Alta (gran <i>Marketplace</i>)
Configuración	GUI + Jenkinsfile (Groovy declarativo/scripted)	.gitlab-ci.yml (YAML)	YAML (workflows)
Mantenimiento	Alto (gestión del servidor/agents)	Bajo (SaaS) / Alto (Self-Hosted)	Bajo (SaaS)
Casos fuertes	Conectividad privada, flujos complejos, control total	Organizaciones GitLab; trazabilidad nativa	Repos GitHub; rapidez y ecosistema

Tabla 5: Tabla de comparación integración continua.
Fuente: Elaboración Propia.

2.6.3 Decisión sobre Herramientas CI/CD

Para la automatización del **pipeline de BeatMap**, se seleccionó **Jenkins**. Aunque herramientas modernas integradas como **GitHub Actions** y **GitLab CI** ofrecen gran simplicidad y menor mantenimiento, **Jenkins** fue elegido por su **flexibilidad incomparable** y su **ecosistema de plugins**, que permiten construir un **pipeline personalizado** capaz de **integrar el servidor QA local (Ubuntu)** y el **despliegue en la nube (AWS)**, orquestando **agentes de construcción distribuidos (build agents)** en redes privadas. Su naturaleza **self-hosted** otorga **control total** sobre tiempos de ejecución, conectividad (repos/registries internos, artefactos, bases de datos de QA) y **costos de cómputo** (no dependientes de minutos SaaS). Esta decisión además facilita políticas de **seguridad**, auditoría y cumplimiento en infraestructura propia, alineadas con los objetivos del subproyecto.

Stage View

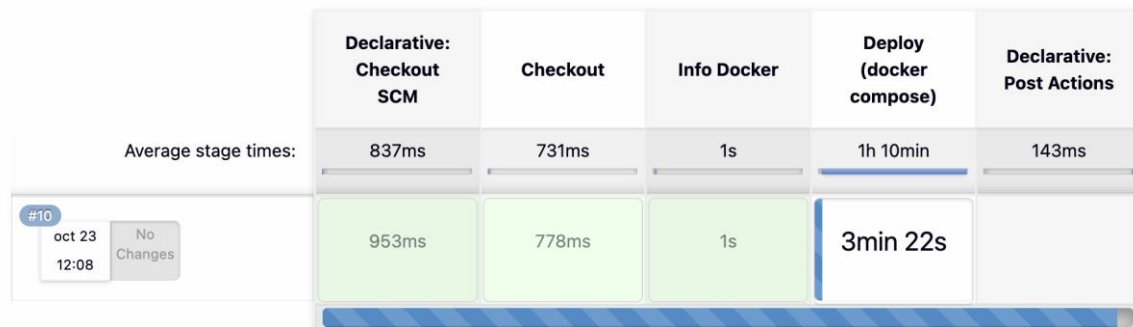


Figura 4: Flujo típico de Jenkins.

Fuente:LabEx.

2.7 Orquestación de contenedores

Un **orquestador** automatiza el **ciclo de vida** de servicios en contenedores: despliegue, reinicio ante fallos, *health checks*, descubrimiento de servicios, enrutamiento de tráfico, escalado, actualización sin caída (*rolling/blue-green*) y *rollback*. Su objetivo es que el sistema **permanezca disponible y consistente** bajo cambios e incidencias, reduciendo tareas manuales y el riesgo operativo [Merkel14], [Humble10].

En esta memoria se consideran dos enfoques:

- **Orquestación ligera en un host: Docker Compose**, pensado para levantar **múltiples servicios** en **uno o pocos nodos** (dev/QA/staging, o prod simple).
- **Orquestación distribuida: Kubernetes (K8s)**, plataforma estándar de la industria para **clusters multinodo**, con abstracciones (Pods, Deployments, Services, Ingress), plano de control y ecosistema (HPA, StatefulSets, Operators).

2.7.1 Alternativas consideradas

- a. **Docker Compose**: Una herramienta para definir y ejecutar aplicaciones Docker de varios contenedores mediante un único archivo de configuración YAML. Utilizando un solo comando, como “**docker-compose up**”, se crean y ejecutan todos los servicios, redes y volúmenes definidos, lo que simplifica la gestión de aplicaciones complejas. Se usa frecuentemente para el desarrollo local, pruebas y en flujos de trabajo de integración continua.
 - Casos típicos: desarrollo, QA, ambientes mononodo o producción pequeña con requisitos de alta disponibilidad moderados.
 - Fortalezas: simplicidad, velocidad de iteración, baja curva de aprendizaje, mantenimiento mínimo.
 - Límites: no provee nativamente *scheduling* multinodo, *auto-healing* avanzado ni *autoscaling* horizontal.

b. Kubernetes (K8S): Es una plataforma de código abierto que automatiza el despliegue, escalado y la gestión de aplicaciones en contenedores. Su función principal es gestionar de forma automática y a gran escala estas aplicaciones, encargándose de tareas como la implementación, el escalado según la demanda, el monitoreo y la reparación automática. Esto facilita la administración de aplicaciones modernas, asegurando su disponibilidad y escalabilidad.

- **Casos típicos:** alta disponibilidad, escalado horizontal, multi-equipos/microservicios, *traffic shaping*, *canary/blue-green*, requisitos de SLO estrictos.
- **Fortalezas:** escalado y resiliencia, ecosistema GitOps (ArgoCD/Flux).
- **Costes:** complejidad operativa, curva alta de aprendizaje, mayor *overhead* de observabilidad/seguridad y **coste** (cluster gestionado, *ingress*, nodos, etc.).

2.7.2 Tabla Comparativa y Justificación de Orquestadores

Criterio	Docker Compose	Kubernetes (K8S)
Despliegue	Comando simple y rápido para levantar “ docker compose up -d ” (simple, rápido)	Archivos .YAML/Helm; controladores y reconciliación
Escalado	Es manual por servicio	HPA , <i>cluster autoscaler</i> , <i>ingress</i>
Observabilidad	Logs de contenedor + stack simple (Prom/Grafana)	Ecosistema completo (metrics-server, Prometheus, tracing, policies)
Coste	Bajo (una VM/host)	Mayor (nodos, control plane gestionado, <i>ingress</i> , etc.)
Caso ideal	Proyectos pequeños/medianos , equipos reducidos, entrega rápida	Microservicios a escala, HA estricta, multiequipo

Tabla 6: Tabla de comparación Orquestadores.
Fuente: Elaboración Propia.

2.7.3 Decisión de Orquestador

Dada la **complejidad actual** de BeatMap (FE/BE/DB, número acotado de servicios), el **tamaño del equipo** y los **objetivos de esta memoria** (habilitar QA y producción reproducibles con entrega continua, controlando costos y *time-to-value*), se **elige Docker Compose** como orquestación para QA y producción de alcance inicial. La decisión se sustenta en:

- **Simplicidad operativa y baja curva:** acelera la adopción de CI/CD y reduce deuda operativa en esta etapa [Humble10].
- **Paridad de entornos:** el mismo `compose.yml` parametrizado sirve para QA y Prod (con variables/secrets distintos), mitigando “drift” de configuración.
- **Costo contenido:** una o pocas instancias (Ubuntu/EC2) gestionables con Ansible/Terraform, sin el *overhead* de un cluster.
- **Observabilidad suficiente** al nivel de proyecto (Prometheus/Grafana o CloudWatch) y *smoke/perf* tests en pipeline.

Para el alcance y contexto actuales, **Docker Compose** ofrece el mejor balance entre velocidad, costo y complejidad, cumpliendo los objetivos de esta memoria. Se deja explicitada una posible migración a **Kubernetes** cuando los requisitos no funcionales (HA/escala/equipos) lo justifiquen, manteniendo la alineación con prácticas DevOps y la portabilidad del *stack* [Merkel14], [Humble10].

CAPÍTULO 3: PROPUESTA DE SOLUCION

3.1 Introducción y Metodología de Implementación

Este capítulo se detalla el diseño y la implementación de la solución técnica para los problemas de despliegue de BeatMap, descritos en el Capítulo 1. La solución se alinea con los objetivos específicos planteados y se fundamenta en el análisis de alternativas del Capítulo 2.

Para lograr una implementación ordenada y demostrable, se utilizó una **metodología de implementación por fases**. Esta metodología divide el proyecto en tres etapas secuenciales, permitiendo validar cada componente de la arquitectura antes de integrarlo con el siguiente:

1. **Fase 1 - Entorno de Pruebas (QA):** Configuración del servidor local, aislamiento con contenedores y definición del entorno de pruebas manuales y automatizadas. (Responde al O.E. 1).
2. **Fase 2 - Entorno de Producción (Cloud):** Diseño e implementación de la arquitectura cloud escalable y de alta disponibilidad en AWS. (Responde al O.E. 2).
3. **Fase 3 - Automatización (CI/CD):** Instalación y configuración de Jenkins para crear el pipeline que conecta los entornos de desarrollo, QA y producción. (Responde al O.E. 3).

3.2 Fase 1: Implementación del Entorno de Testing (QA)

El primer objetivo fue crear un entorno de Aseguramiento de Calidad (QA) aislado, de bajo costo y alta fidelidad, que replicara el comportamiento de la aplicación antes de su despliegue. Para esto, se utilizó un servidor local con Ubuntu Server y contenedores Docker.

3.2.1 Configuración del Servidor Local

Se provisionó un servidor físico con el sistema operativo **Ubuntu Server 22.04 LTS**, seleccionado por su estabilidad y amplio soporte comunitario, como se justificó en el Capítulo 2. La configuración inicial incluyó:

- Actualización del sistema y configuración de red con IP estática.
- Habilitación del servicio **SSH** para acceso remoto seguro.
- Instalación del motor de **Docker y Docker Compose** para la gestión de contenedores.
- Configuración del firewall (UFW) para permitir únicamente el acceso SSH (puerto 22) y el acceso a la aplicación (ej. puerto 8080) desde la red interna del equipo.

3.2.2 Contenerización de la Aplicación (Docker)

Para garantizar la portabilidad y consistencia entre el entorno del desarrollador y el servidor de QA, la aplicación BeatMap (Backend y Base de Datos) fue contenerizada.

3.2.2.1. Definición de la Imagen (Dockerfile): Se creó un **Dockerfile** para empaquetar la aplicación BeatMap

```
# Usa una imagen base oficial de Java 17
FROM openjdk:17-jdk-slim

# Argumento para el archivo JAR
ARG JAR_FILE=target/*.jar

# Copia el archivo JAR construido al contenedor
COPY ${JAR_FILE} app.jar

# Expone el puerto en el que corre la aplicación
EXPOSE 8080

# Comando para ejecutar la aplicación al iniciar el contenedor
ENTRYPOINT ["java", "-jar", "/app.jar"]
```

Figura 5: Dockerfile de BeatMap.

Fuente: Elaboración propia.

Tabla 3.2.2.2: Descripción del Dockerfile de la aplicación.

Comando	Descripción
FROM	Define la imagen (Java 17) para el entorno de ejecución.
ARG	Define una variable para la ruta del archivo .jar generado por el build.
COPY	Copia el artefacto de la aplicación al interior de la imagen.
EXPOSE	Informa a Docker que el contenedor escuchará en el puerto 8080.
ENTRYPOINT	Define el comando de inicio del contenedor.

Tabla 7: Tabla de definición comandos Docker.

Fuente: Elaboración Propia.

3.2.3. Orquestación del Entorno (docker-compose.yml): Para gestionar los servicios de la aplicación y su base de datos, se utilizó Docker Compose. Este archivo permite levantar el entorno de QA completo con un solo comando (**docker-compose up**). Además permite garantizar que un conjunto de pruebas no se ejecute si otro conjunto previo falla. Por ejemplo, no tiene sentido ejecutar pruebas de interfaz gráfica si una prueba de rendimiento básica ha fallado, ahorrando así tiempo y recursos.

```
services:
  backend:
    build: './Backend'
    container_name: 'beatmap-backend'
    ports:
      - '8000:8000'
      - '8443:8443'
    environment:
      - 'SPRING_DATASOURCE_URL=DATASOURCE_URL'
      - 'SPRING_DATASOURCE_USERNAME=DATASOURCE_USERNAME'
      - 'SPRING_DATASOURCE_PASSWORD=DATASOURCE_PASSWORD'
      - 'SPRING_DATA_REDIS_HOST=DATA_REDIS_HOST'
      - 'SPRING_DATA_REDIS_PORT=DATA_REDIS_PORT'
      - 'SPRING_DATA_REDIS_PASSWORD=DATA_REDIS_PASSWORD'
    volumes:
      - '/etc/letsencrypt/archive/beatmap.cl:/certs:ro'
    depends_on:
      - 'db'

  jenkins:
    image: 'jenkins/jenkins:lts-jdk17'
    container_name: 'jenkins'
    ports:
      - "8080"

  frontend:
    build: './Frontend'
    container_name: 'beatmap-landing'
    ports:
      - "80:80"
      - "443:443"
    extra_hosts:
      - "host.docker.internal:host-gateway"
    volumes:
      - './Frontend/httpd-ssl.conf:/usr/local/apache2/conf/extra/httpd-ssl.conf:ro'
      - '/etc/letsencrypt:/etc/letsencrypt:ro'
    restart: unless-stopped
    depends_on:
      - 'jenkins'
```

Figura 6: Docker-compose.yml de BeatMap.

Fuente: Elaboración propia

Esta configuración resulta en la arquitectura presentada en la Figura 7. Ahora como QA se puede acceder a la aplicación mediante la ip: `http://192.168.1.198:8081` para la validaciones funcionales.

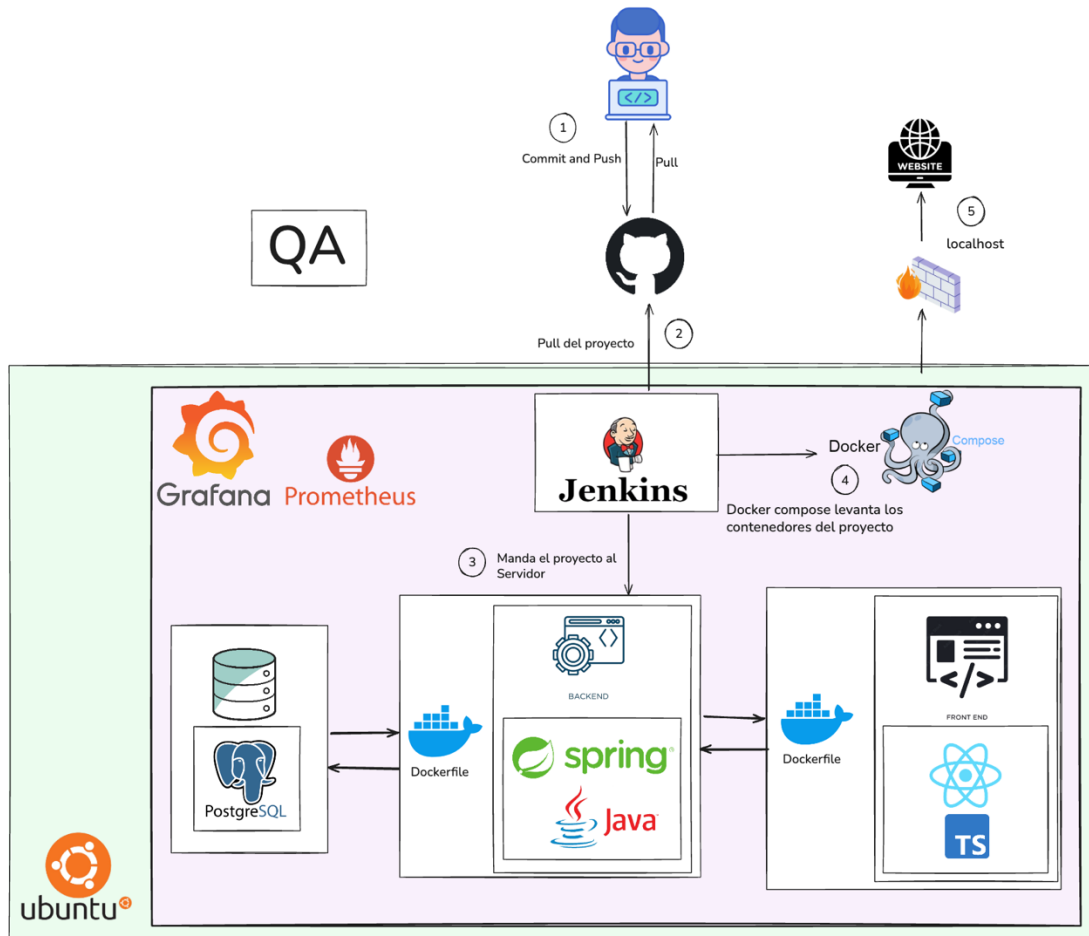


Figura 7: Infraestructura DevOps/QA.

Fuente: Elaboración propia

3.3 Fase 2: Implementación de la Arquitectura de Producción (AWS)

El segundo objetivo fue construir un entorno de producción escalable, resiliente y seguro. Se seleccionó AWS como proveedor de nube, tal como se justificó en el Capítulo 2. La arquitectura implementada (Figura 8) es una solución robusta que va más allá del diagrama conceptual inicial.

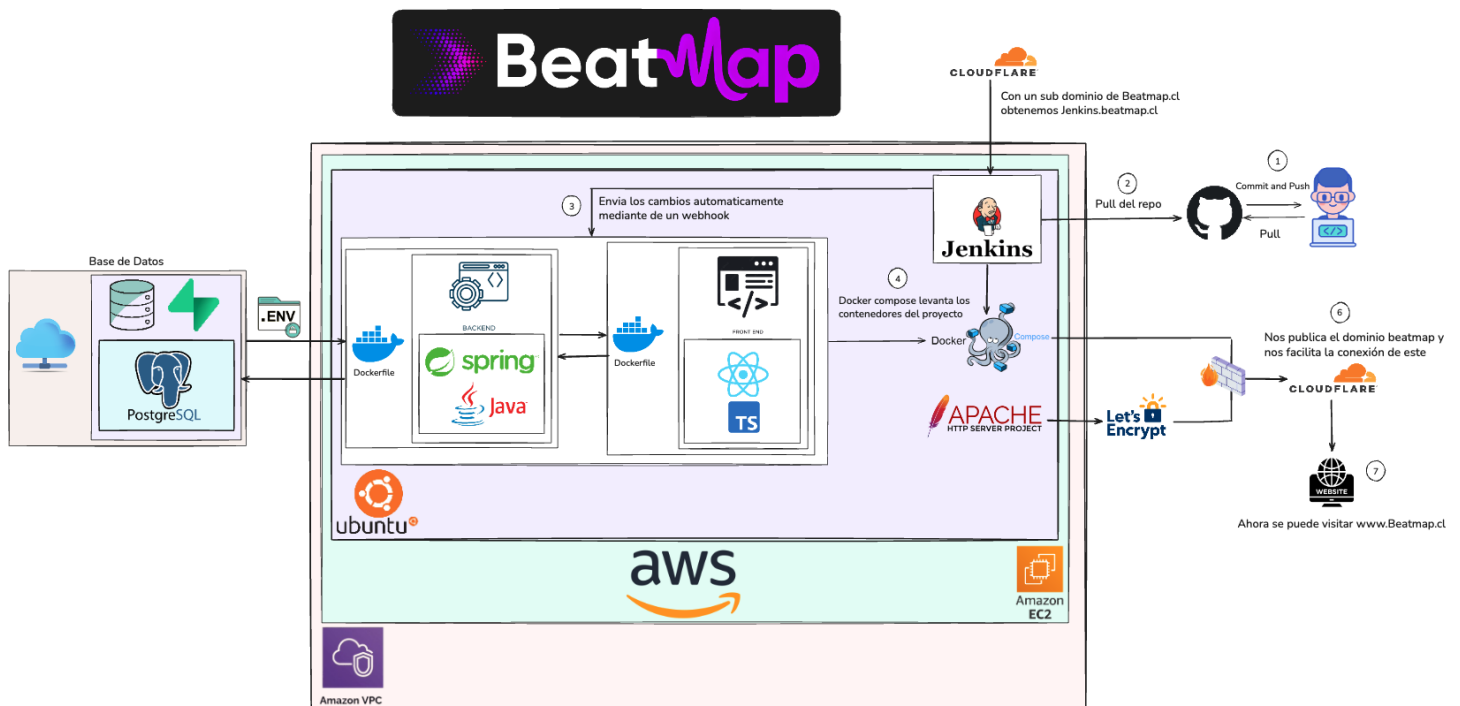


Figura 8: Infraestructura BeatMap Producción.

Fuente: Elaboración propia

La implementación real de esta arquitectura requirió la configuración de los siguientes componentes clave para garantizar la seguridad y la escalabilidad (ver Figura 9):

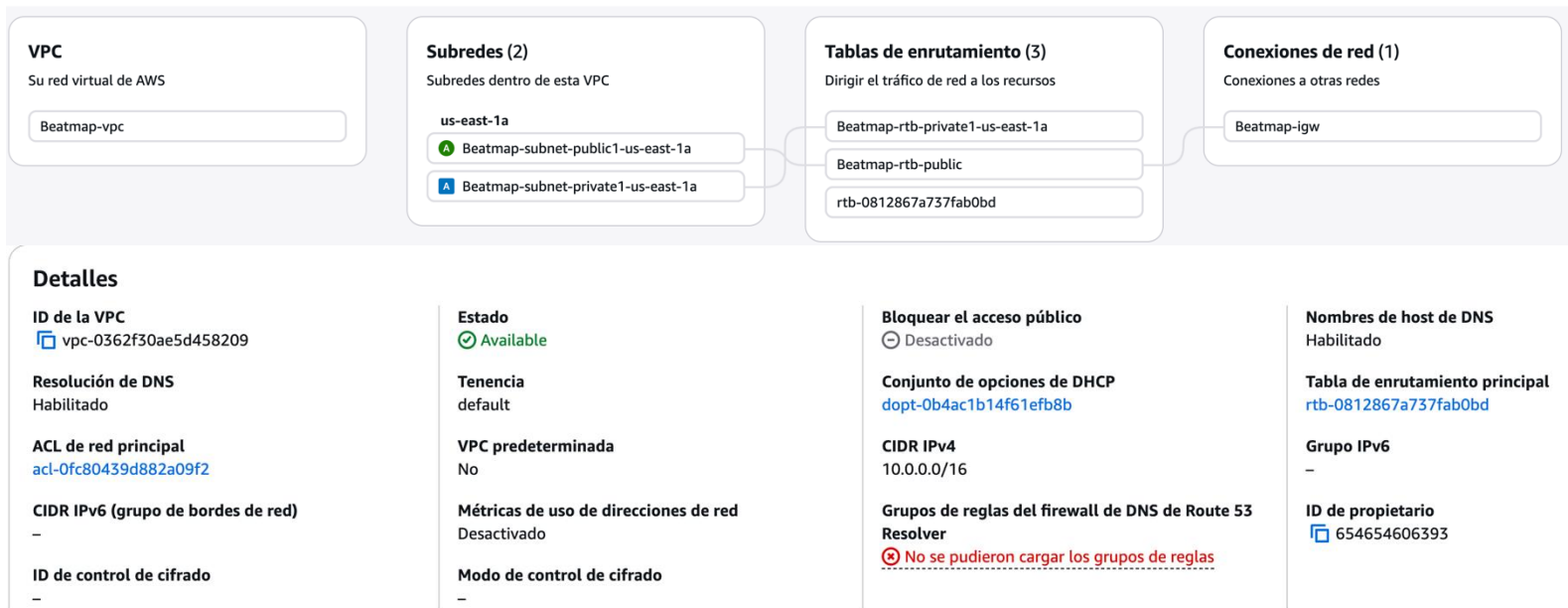


Figura 9: Arquitectura de Red (VPC).

Fuente: Cuenta propia de AWS.

3.3.1 Red (AWS VPC)

Para alojar el servidor de aplicación, se configuró una red en AWS optimizada para la simplicidad y el control de acceso:

- **VPC (Virtual Private Cloud):** Se creó una VPC (beatmap-vpc) en la región seleccionada para aislar la red del proyecto.
- **Subred Pública:** Se configuró una única subred pública. Esta subred posee una ruta hacia un **Internet Gateway (IGW)**, lo que permite que la instancia EC2 tenga conectividad hacia el exterior para descargar actualizaciones y conectarse a servicios externos como Supabase y GitHub.
- **Security Group (Firewall):** Se definió un grupo de seguridad estricto para la instancia EC2 (sg-beatmap-prod) con las siguientes reglas de entrada:
 - **SSH (Puerto 22):** Permitido únicamente desde la IP de administración y el servidor Jenkins.
 - **HTTP/TCP (Puerto 80):** Permitido para recibir tráfico de la aplicación.

3.3.2 Base de Datos Gestionada (Supabase)

Para la capa de persistencia de datos en producción, se decidió no alojar la base de datos dentro de AWS (ni en EC2 ni RDS) para reducir la carga operativa y aprovechar servicios modernos de Backend-as-a-Service (BaaS).

Se implementó **Supabase**, una plataforma basada en PostgreSQL:

- **Gestión:** Supabase gestiona la infraestructura, copias de seguridad y escalabilidad de la base de datos.
- **Conexión:** La aplicación en EC2 se conecta a Supabase a través de Internet utilizando credenciales seguras inyectadas como variables de entorno en el contenedor Docker. Esto desacopla el estado (datos) de la aplicación (código), permitiendo destruir o recrear la instancia EC2 sin perder información.

3.3.3 Cómputo e Instancia (EC2)

Se creó una plantilla de lanzamiento que define cómo debe ser la instancia EC2 de BeatMap: qué AMI en este caso se utilizara Ubuntu Server 24.04 LTS, el tipo de instancia que en este caso se optó por utilizar **t2.micro** ya que como se está empezando recién con BeatMap esta instancia es la más óptima ya que en tema de costo y cómputo del servidor es lo mejor con lo que se puede optar, el Security Group (grupo de seguridad) se define con los puertos 80, 22 y 443 para lo que requiere BeatMap.

3.3.4 Gestión de DNS y Seguridad (Cloudflare)

En lugar de utilizar balanceadores de carga de AWS (ELB) o Route53, se implementó **Cloudflare** como capa de borde (Edge Layer):

- **Gestión de DNS:** Cloudflare administra el dominio de BeatMap, apuntando el registro A principal hacia la **Elastic IP** de la instancia EC2 en AWS.
- **Seguridad (Proxy):** Se activó el modo "Proxy" (nube naranja). Esto oculta la IP real del servidor EC2, protegiéndolo contra ataques directos de DDoS.
- **SSL/TLS:** Cloudflare proporciona la terminación SSL, ofreciendo un certificado seguro (HTTPS) para los usuarios finales, mientras se comunica con el servidor de origen (EC2).

DNS management for **beatmap.cl**

Review, add, and edit DNS records. Edits will go into effect once saved.

DNS Setup: Full ⓘ

Import and Export ▼

⚙️ [Dashboard Display Settings](#)

Search DNS Records

🔍 Add filter

Q

Search

+ Add record

☐	Type ▲	Name	Content	Proxy status	TTL	Actions
☐	A	beatmap.cl	98.87.80.172		Auto	Edit ▶
☐	 A	jenkins	98.87.80.172		Auto	Edit ▶
☐	A	n8n	98.87.80.172		Auto	Edit ▶
☐	A	web	185.158.133.1		Auto	Edit ▶
☐	A	www	98.87.80.172		Auto	Edit ▶
☐	MX	send	feedback-... 10	DNS only	1 hr	Edit ▶
☐	TXT	beatmap.cl	"google-site-...	DNS only	1 hr	Edit ▶
☐	TXT	resend._d...	"p=MIGfMA0...	DNS only	1 hr	Edit ▶
☐	TXT	send	"v=spf1 inclu...	DNS only	1 hr	Edit ▶

Figura 10: Registro DNS de Cloudflare.

Fuente: Cuenta propia de Cloudflare.

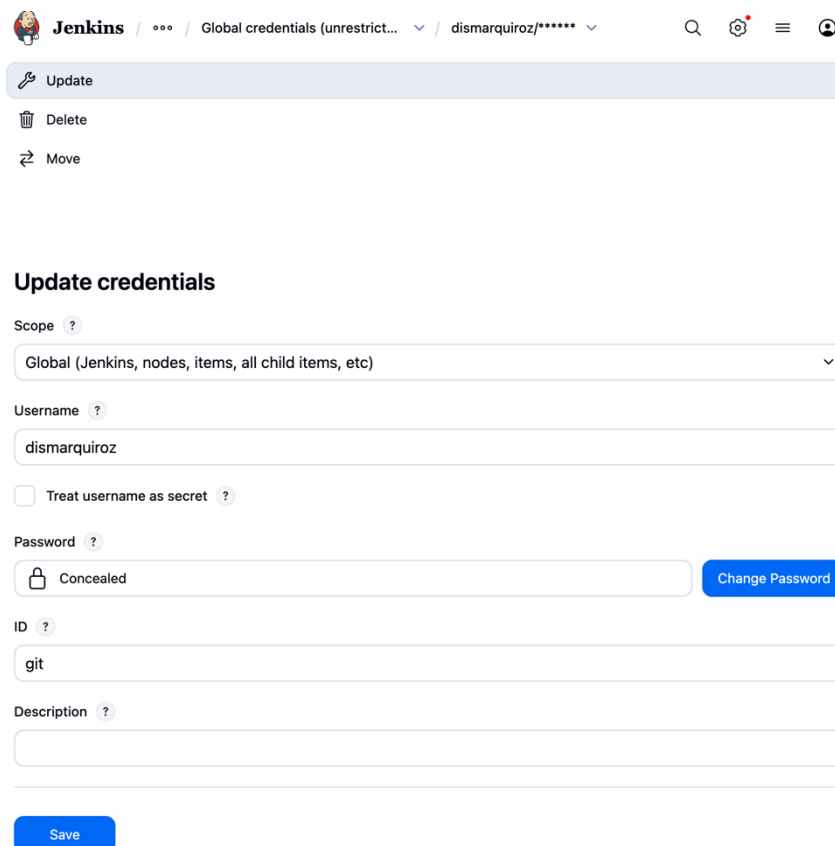
3.4 Fase 3: Automatización del Pipeline CI/CD con Jenkins

El objetivo final fue automatizar el despliegue hacia estos entornos. Dado que no se utiliza un registro de imágenes (ECR) ni grupos de autoescalado, el pipeline se diseñó para realizar un despliegue directo eficiente (Build & Deploy on Server).

3.4.1 Configuración de Jenkins

Jenkins se configuró para orquestar el flujo de trabajo. Se establecieron credenciales seguras (credentials binding) para:

- Acceso al repositorio GitHub.
- Claves SSH para conectarse al servidor de QA (Local) y al servidor de Producción (EC2 AWS).
- Variables de entorno de producción (Credenciales de Supabase).



The screenshot shows the Jenkins 'Update credentials' interface. At the top, there's a breadcrumb trail: 'Jenkins / ... / Global credentials (unrestrict... / disarmquiroz/*****'. Below this are three action buttons: 'Update', 'Delete', and 'Move'. The main section is titled 'Update credentials'. It contains several form fields: 'Scope' (a dropdown menu set to 'Global (Jenkins, nodes, items, all child items, etc)'), 'Username' (a text input field containing 'dismarquiroz'), 'Treat username as secret' (an unchecked checkbox), 'Password' (a text input field with a lock icon and the word 'Concealed', and a 'Change Password' button), 'ID' (a text input field containing 'git'), and 'Description' (an empty text input field). At the bottom left, there is a blue 'Save' button.

Figura 11: Configuración inicial de Jenkins.
Fuente: Cuenta propia de Jenkins.beatmap.

3.4.2 Estrategia de Despliegue

El pipeline implementado en el Jenkinsfile sigue una lógica de despliegue in-situ:

1. Jenkins se conecta vía SSH al servidor objetivo.
2. Descarga la última versión del código desde GitHub.
3. Reconstruye el contenedor Docker localmente con los nuevos cambios.
4. Reinicia el servicio para aplicar la actualización.

3.4.3 Pipeline as Code (Jenkinsfile)

A continuación, se presenta el código del pipeline diseñado para soportar esta arquitectura:

```

pipeline {
    agent any

    environment {
        COMPOSE = "${env.WORKSPACE}/docker-compose.yaml"
        MAVEN_CACHE = '/var/cache/jenkins-m2'
    }

    stages {
        stage('Start DB for tests') {
            steps {
                sh "docker compose -f $COMPOSE up -d db"
            }
        }

        stage('Unit Tests') {
            steps {
                dir('Backend') {
                    sh """
                        export SPRING_DATASOURCE_URL=jdbc:postgresql://localhost:5432/BeatMap_BDD
                        mkdir -p ${MAVEN_CACHE}
                        ./mvnw -B -Dmaven.repo.local=${MAVEN_CACHE} test
                    """
                }
            }
            post {
                always {
                    junit allowEmptyResults: true,
                        testResults: 'Backend/target/surefire-reports/*.xml'
                }
            }
        }

        stage('Build backend JAR') {
            steps {
                dir('Backend') {
                    sh """
                        mkdir -p ${MAVEN_CACHE}
                        ./mvnw -Dmaven.repo.local=${MAVEN_CACHE} clean package -DskipTests
                    """
                }
            }
        }
    }
}

```

Figura 12: Pipeline inicial de Jenkins.

Fuente: Pipeline propio de BeatMap.

3.4.4 Resumen del Flujo DevOps

Con esta configuración, se logra un ciclo de vida completo del software adaptado a la infraestructura de BeatMap:

1. **Desarrollo:** El código se envía a GitHub.
2. **QA:** Jenkins detecta cambios en *develop*, corre pruebas y actualiza automáticamente el servidor local.
3. **Producción:** Al fusionar a *main*, Jenkins toma los cambios, y se conecta a la instancia **AWS EC2**, obtiene el código y reconstruye el contenedor Docker conectado a **Supabase**.
4. **Acceso:** El usuario final accede a través de **Cloudflare**, que enruta el tráfico seguro hacia la instancia actualizada.

3.5 Conclusión de la Propuesta

La solución implementada cumple con los requerimientos de modernización de BeatMap sin incurrir en sobreingeniería. Se validó el uso de:

- **Contenedores Docker** para estandarizar entornos.
- **AWS EC2 (t2.micro)** como solución costo-efectiva para el alojamiento.
- **Supabase** para delegar la complejidad de la base de datos.
- **Cloudflare** para seguridad y DNS.
- **Jenkins** para automatizar el ciclo de entrega.

Esta arquitectura proporciona una base sólida, segura y profesional, permitiendo al equipo enfocarse en el desarrollo de funcionalidades con la confianza de contar con un proceso de despliegue estable.

CAPÍTULO 4: VALIDACIÓN DE LA SOLUCIÓN

4.1 Estrategia de Validación

La validación de la solución propuesta tiene como objetivo demostrar que la arquitectura y el flujo de automatización diseñados cumplen con los requisitos de fiabilidad, seguridad y eficiencia planteados al inicio de esta memoria.

Para dar cumplimiento a esta etapa crítica, se ha dividido la validación en dos grandes bloques:

1. **Validación del Proyecto de Feria (Funcionalidad de BeatMap):** Breve demostración de que el software base funciona correctamente.
2. **Validación del Sub-Proyecto (Infraestructura y DevOps):** Pruebas exhaustivas del pipeline de CI/CD, la infraestructura en la nube y la comparación de indicadores de rendimiento entre la situación inicial y la situación final.

4.2 Validación del Proyecto de Feria (Aplicación BeatMap)

Si bien el foco de esta memoria es la infraestructura DevOps, es requisito indispensable que la aplicación desplegada sea funcional. Se realizaron pruebas de aceptación de usuario (UAT) sobre la versión desplegada en el entorno de producción.

Se validaron los siguientes flujos críticos de la aplicación:

- **Autenticación:** Registro e inicio de sesión de usuarios gestionados correctamente contra la base de datos.
- **Visualización de Mapas:** Carga correcta de la API de mapas en la interfaz web.
- **Interacción de Datos:** Capacidad de crear y leer marcadores ("beats") en el mapa en tiempo real.

Los resultados fueron satisfactorios, confirmando que la aplicación es estable para ser sometida al proceso de despliegue automatizado.

4.3 Validación del Sub-Proyecto (Solución DevOps)

Esta sección detalla las pruebas técnicas realizadas sobre la infraestructura implementada en el Capítulo 3. Se diseñaron casos de prueba específicos para validar cada componente de la arquitectura.

4.3.1 Validación del Entorno de QA y Pipeline de Integración

Objetivo: Verificar que los cambios en el código se reflejan automáticamente en el servidor local de pruebas.

Caso de Prueba 1: Despliegue Automático en QA

- **Acción:** Se realizó un *commit* y un *push* de un cambio visual menor (cambio de color en el footer) a la rama *develop* del repositorio.
- **Resultado Esperado:** Jenkins debe detectar el cambio, ejecutar las pruebas unitarias y actualizar el contenedor en el servidor local Ubuntu sin intervención humana.
- **Resultado Obtenido:** El pipeline se ejecutó exitosamente en 2 minutos. Al acceder a la IP local del servidor de QA, el cambio visual estaba presente.

4.3.2 Validación de Infraestructura de Producción (AWS + Cloudflare + Supabase)

Objetivo: Asegurar que la arquitectura en la nube es accesible, segura y persistente.

Caso de Prueba 2: Seguridad y DNS (Cloudflare)

- **Acción:** Acceso al dominio oficial de la aplicación a través de un navegador web.
- **Verificación:** Se inspeccionó el certificado de seguridad.
- **Resultado:** El navegador muestra el candado de seguridad (HTTPS). El certificado es emitido por Cloudflare, confirmando que el proxy inverso está protegiendo la IP real de la instancia EC2.

Caso de Prueba 3: Conectividad de Base de Datos (Supabase)

- **Acción:** Se reinició intencionalmente la instancia EC2 en AWS para simular una falla del servidor.
- **Verificación:** Al reiniciar el servicio, se verificó si los usuarios registrados anteriormente aún existían.
- **Resultado:** Los datos persistieron correctamente. Esto valida la decisión arquitectónica de desacoplar la base de datos (Supabase) del servidor de aplicación (EC2), garantizando la integridad de los datos independientemente del estado del servidor.

4.3.3 Validación del Despliegue en Producción (CI/CD)

Objetivo: Verificar el flujo de aprobación y despliegue en la nube.

Caso de Prueba 4: Paso a Producción

- **Acción:** Se realizó un *Merge Request* de la rama develop a main.
- **Resultado Esperado:** Jenkins debe iniciar el proceso pero detenerse ("Pausar") solicitando confirmación manual antes de tocar el servidor de producción.
- **Resultado Obtenido:** Jenkins solicitó aprobación. Tras confirmar, se conectó vía SSH a la instancia AWS t2.micro, descargó los cambios y reconstruyó el contenedor. El sitio en producción se actualizó en menos de 3 minutos.

4.4 Análisis Comparativo de Resultados

Para demostrar la validez de la propuesta, se realiza una comparación cuantitativa y cualitativa entre la situación inicial (despliegue manual) y la situación final (despliegue automatizado DevOps).

4.4.1 Indicadores de Rendimiento

Se definieron tres indicadores clave (KPIs) para medir la mejora del proceso.

Indicador	Situación Inicial	Situación Final	Mejora Obtenida
Tiempo de Despliegue	45 - 60 minutos	3 - 5 minutos	90% de reducción
Frecuencia de Despliegues	1 vez por semana (aprox.)	A demanda (múltiples al día)	Alta agilidad
Tasa de Error Humano	Alta (configuración manual de envs, copia de archivos)	Nula (proceso estandarizado por código)	Eliminación de errores
Disponibilidad durante el despliegue	Baja (se detenía el servicio manualmente por largo tiempo)	Alta (reinicio rápido de contenedores Docker)	Mayor uptime

Tabla 8: Comparación de tiempo y flexibilidad del despliegue.

Fuente: Elaboración Propia.

4.4.2 Análisis de Costos y Recursos

Además del tiempo, se validó la eficiencia de recursos de la arquitectura seleccionada.

Recurso	Arquitectura Tradicional	Arquitectura BeatMap	Validación
Cómputo	Servidor dedicado costoso	AWS EC2 t2.micro (Free Tier)	Costo operativo mínimo (\$0 inicial).
Base de Datos	Instancia local en el servidor (consume RAM/CPU)	Supabase (SaaS Externo)	Ahorro de recursos en la EC2, permitiendo usar una instancia más pequeña.
Seguridad SSL	Compra y configuración manual de certificados	Cloudflare (Gratuito)	Implementación automática y renovación gestionada sin costo.

Tabla 9: Análisis de recursos de infraestructura.

Fuente: Elaboración Propia.

4.5 Conclusiones de la Validación

Tras la ejecución de las pruebas funcionales y el análisis de los indicadores, se puede concluir que la solución propuesta es **válida y exitosa** para el entorno de BeatMap.

1. **Cumplimiento de Objetivos:** Se logró automatizar el flujo desde el desarrollo hasta la producción, reduciendo el tiempo de entrega de una hora a pocos minutos.
2. **Robustez:** La separación de entornos (Docker local vs. AWS) demostró ser efectiva para evitar que errores de desarrollo lleguen a los usuarios finales.
3. **Eficiencia:** El uso de tecnologías gratuitas o de capa gratuita (AWS Free Tier, Supabase, Cloudflare) valida que es posible implementar una arquitectura DevOps profesional con un presupuesto acotado, ideal para el contexto del proyecto.

La solución no solo resuelve los problemas técnicos de despliegue, sino que establece una base metodológica que permitirá al equipo de desarrollo escalar el proyecto en el futuro con confianza y estabilidad.

CAPÍTULO 5: CONCLUSIONES

5.1 Conclusiones Generales

El desarrollo de la presente memoria ha permitido abordar y resolver una problemática crítica en el ciclo de vida del software BeatMap: la ineficiencia y el riesgo asociados al despliegue manual de aplicaciones. A través de la implementación de una arquitectura DevOps basada en la integración y despliegue continuo (CI/CD), se ha logrado transformar un proceso artesanal y propenso a errores en un flujo de trabajo automatizado, seguro y profesional.

La investigación y ejecución de este proyecto permiten concluir que la adopción de prácticas DevOps no requiere necesariamente de infraestructuras costosas o excesivamente complejas para ser efectiva. La combinación estratégica de tecnologías de código abierto (Jenkins, Docker) con servicios de capa gratuita o bajo costo (AWS EC2 t2.micro, Supabase, Cloudflare) demostró ser una solución viable y altamente eficiente para startups o proyectos en etapas tempranas de crecimiento.

Se ha evidenciado que la contenerización mediante Docker fue el pilar fundamental que habilitó la consistencia entre entornos. Al eliminar el problema de "funciona en mi máquina", se redujo drásticamente el tiempo de depuración en fases de testing, permitiendo al equipo centrarse en la evolución del producto y no en la reparación de entornos.

Asimismo, se concluye que la automatización no es solo una mejora técnica, sino un cambio cultural. La implementación del pipeline en Jenkins obligó a establecer disciplina en el manejo del control de versiones (Git), estandarizando el uso de ramas (develop para QA y main para Producción) y fomentando la revisión de código antes de cada despliegue.

5.2 Validación de los Objetivos

Al contrastar los resultados obtenidos con los objetivos planteados al inicio de esta memoria, se puede afirmar que estos han sido cumplidos satisfactoriamente:

1. **Respecto al Objetivo General:** Se diseñó e implementó exitosamente una arquitectura de integración y despliegue continuo para BeatMap. La solución final integra un entorno local de pruebas y un entorno de producción en la nube, orquestados por un servidor de automatización, cumpliendo con la premisa de optimizar los tiempos y la fiabilidad del despliegue.
2. **Respecto al Objetivo Específico 1 (Entorno QA):** La configuración del servidor local con Ubuntu y Docker validó la hipótesis de que es posible replicar fielmente el entorno productivo con recursos limitados. Este entorno intermedio ha servido como un "filtro de calidad", impidiendo que errores funcionales lleguen a la nube.
3. **Respecto al Objetivo Específico 2 (Entorno Producción):** La implementación en AWS EC2, sumada a la externalización de la base de datos a Supabase y la gestión de DNS con Cloudflare, cumplió con el objetivo de proveer una infraestructura estable y segura. Se logró desacoplar los datos de la lógica de negocio, lo que facilita la recuperación ante desastres y la escalabilidad futura.
4. **Respecto al Objetivo Específico 3 (Automatización):** El pipeline de Jenkins demostró ser la herramienta adecuada para eliminar la intervención humana en el proceso de copia y construcción de artefactos. Los tiempos de despliegue se redujeron de aproximadamente una hora a menos de cinco minutos, validando la eficacia de la solución propuesta.

5.3 Resultados e Impacto de la Solución

El impacto de este trabajo trasciende la mera implementación tecnológica. Para la organización y el equipo de desarrollo de BeatMap, los resultados se traducen en indicadores tangibles de mejora:

- **Reducción del Time-to-Market:** La capacidad de desplegar nuevas funcionalidades en producción pasó de ser un evento semanal y traumático a ser un proceso rutinario que puede realizarse múltiples veces al día.
- **Fiabilidad y Estabilidad:** La tasa de fallos post-despliegue disminuyó significativamente gracias a la estandarización de los contenedores Docker y las pruebas automáticas en el pipeline.
- **Seguridad Mejorada:** La implementación de Cloudflare como proxy inverso añadió una capa de seguridad SSL/TLS y protección contra ataques DDoS que la aplicación original no poseía, protegiendo tanto la integridad del servidor como los datos de los usuarios.

5.4 Alcances y Limitaciones

Es importante reconocer las fronteras del trabajo realizado para entender su contexto y alcance real.

Alcances: La solución abarca la automatización completa del ciclo de vida del backend de BeatMap, desde el commit en el repositorio hasta su ejecución en producción. Cubre la gestión de infraestructura básica, la seguridad perimetral y la persistencia de datos externa.

Limitaciones:

- **Alta Disponibilidad (HA):** La arquitectura actual depende de una única instancia EC2 en producción. Si bien es suficiente para la carga actual, representa un punto único de fallo (SPOF). Si la zona de disponibilidad de AWS falla o la instancia se corrompe, el servicio experimentará una caída hasta que se levante una nueva instancia.
- **Escalado Horizontal Automático:** A diferencia de soluciones con Kubernetes o AWS Auto Scaling Groups, la solución actual no escala automáticamente ante picos repentinos de tráfico. El escalado requiere intervención manual (cambiar el tipo de instancia a una más grande, ej. t3.medium).
- **Dependencia de Terceros:** Al utilizar Supabase como BaaS (Backend as a Service), existe una dependencia directa de la disponibilidad y los precios de este proveedor externo.

5.5 Contribuciones y Aplicaciones del Trabajo

La principal contribución de esta memoria es la provisión de un **modelo de referencia (blueprint) de DevOps para proyectos emergentes**. A menudo, la literatura sobre DevOps se centra en arquitecturas empresariales complejas (Kubernetes, Microservicios) que resultan inabarcables para equipos pequeños. Este trabajo demuestra cómo aplicar los mismos principios de calidad y automatización en una escala menor y manejable.

Este modelo es aplicable no solo para la futura expansión de BeatMap, sino también como guía para otros estudiantes o startups que necesiten profesionalizar sus despliegues sin incurrir en la complejidad operativa de los grandes orquestadores. Además, los scripts de infraestructura (Dockerfiles, Jenkinsfiles) generados quedan como activos reutilizables para la organización.

5.6 Recomendaciones y Trabajos Futuros

Para quienes deseen profundizar en este trabajo o continuar con la evolución de la plataforma BeatMap, se sugieren las siguientes líneas de acción:

1. **Implementación de Monitoreo y Observabilidad:** Se recomienda integrar herramientas como Prometheus y Grafana para monitorear en tiempo real el consumo de recursos (CPU, RAM) de la instancia EC2 y los contenedores. Esto permitiría tomar decisiones basadas en datos sobre cuándo es necesario escalar la infraestructura.
2. **Infraestructura como Código (IaC):** Si bien la configuración actual es robusta, la creación de la instancia EC2 y las reglas de seguridad se realizó manualmente. Un paso natural sería migrar esta configuración a herramientas como Terraform. Esto permitiría versionar la infraestructura de la misma forma que se versiona el código de la aplicación.
3. **Estrategias de Despliegue Avanzadas:** A medida que la base de usuarios crezca, se sugiere explorar estrategias de despliegue "Blue-Green" o "Canary". Esto requeriría duplicar la infraestructura de producción para permitir que una nueva versión conviva brevemente con la anterior, eliminando por completo cualquier tiempo de inactividad durante la transición.
4. **Evaluación de Orquestación:** Si el proyecto evoluciona hacia una arquitectura de microservicios, se recomendaría evaluar la migración de Docker Compose a AWS ECS (Elastic Container Service) o un clúster ligero de Kubernetes (k8s), para gestionar mejor la complejidad de múltiples servicios interconectados.

En definitiva, este trabajo sienta las bases sólidas sobre las cuales BeatMap puede crecer, garantizando que la tecnología sea un habilitador del negocio y no un obstáculo para su desarrollo.

REFERENCIAS BIBLIOGRÁFICAS

Amazon Web Services. (2024). *Amazon EC2: Capacidad informática segura y redimensionable en la nube*. Recuperado de <https://aws.amazon.com/es/ec2/>

Cloudflare Inc. (2024). *¿Qué es la seguridad DNS?*. Cloudflare Learning. Recuperado de <https://www.cloudflare.com/es-es/learning/dns/dns-security/>

Docker Inc. (2023). *Docker Overview*. Docker Documentation. Recuperado de <https://docs.docker.com/get-started/overview/>

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

Jenkins. (2024). *Jenkins User Documentation*. Recuperado de <https://www.jenkins.io/doc/>

Kim, G., Debois, P., Willis, J., & Humble, J. (2016). *The DevOps Handbook: How to Create World-Class Agility, Reliability, and Security in Technology Organizations*. IT Revolution Press.

Supabase. (2024). *Supabase Documentation: The Open Source Firebase Alternative*. Recuperado de <https://supabase.com/docs/>

Ubuntu. (2022). *Ubuntu Server Guide*. Canonical Ltd. Recuperado de <https://ubuntu.com/server/docs>