1

Implementación de Pipeline CI/CD con Kubernetes, Docker y GitHub Actions para Sistemas Distribuidos

Ing. Omar Sánchez Gudiño *(Universidad de Guadalajara, omar.gudino@alumnos.udg.mx)*

Se implementó un pipeline completo de CI/CD utilizando Docker, Kubernetes y GitHub Actions para automatizar el ciclo de desarrollo de software. La solución consistió en containerizar la aplicación con Docker, orquestar los contenedores mediante Kubernetes en Minikube, y automatizar los procesos de build, test y deployment con GitHub Actions. Los contenedores se almacenaron en GitHub Container Registry (GHCR). El pipeline demostró funcionamiento exitoso, ejecutándose automáticamente con cada commit y generando métricas de rendimiento. Esta implementación establece una base sólida para despliegues confiables y escalables, representando una práctica esencial de DevOps para el desarrollo moderno de software distribuido.

INTRODUCCIÓN

El desarrollo de software moderno requiere procesos eficientes que aseguren la calidad, consistencia y confiabilidad de las aplicaciones en producción. Este documento presenta la implementación de un pipeline completo de CI/CD (Integración Continua/Despliegue Continuo) diseñado para automatizar el ciclo de desarrollo de un proyecto de behavioral cloning, utilizando las mejores prácticas de DevOps.

La implementación se centra en cuatro tecnologías fundamentales: Docker para la containerización de la aplicación, Kubernetes para la orquestación de contenedores mediante Minikube, GitHub Actions para la automatización del pipeline CI/CD, y GitHub Container Registry (GHCR) como repositorio privado de imágenes Docker. Esta arquitectura permite gestionar de manera eficiente el código almacenado en el repositorio GitHub "OmarGudi/Proyect_Thesis", ubicado en la ruta local "C:\Users\omarg\OneDrive\Documentos\Proyect_Thesis".

El objetivo principal de este trabajo es establecer un flujo de desarrollo automatizado que, ante cada modificación en la rama principal del repositorio, ejecute automáticamente procesos de build, testing y deployment, garantizando así la entrega consistente y confiable de la aplicación. Esta implementación no solo optimiza el proceso de desarrollo, sino que también sienta las bases para un sistema escalable y mantenible, esencial para proyectos de machine learning como el behavioral cloning donde la reproducibilidad y consistencia son críticas.

La relevancia de este proyecto radica en su aplicabilidad directa a entornos productivos, demostrando cómo las prácticas de DevOps pueden integrarse efectivamente en proyectos de inteligencia artificial para mejorar la calidad del software y acelerar los ciclos de entrega.

MARCO TEORICO

2.1 Fundamentos de DevOps y CI/CD

DevOps representa una evolución cultural y técnica en el desarrollo de software que busca romper las barreras tradicionales entre equipos de desarrollo y operaciones. Esta metodología se sustenta en tres principios fundamentales: flujo de trabajo continuo, retroalimentación constante y aprendizaje experimental. La Integración Continua y Despliegue Continuo (CI/CD) constituye la columna vertebral de DevOps, permitiendo la automatización de procesos desde el commit de código hasta el despliegue en producción.

El CI/CD moderno se conceptualiza como un pipeline automatizado que incluye stages de build, test, security scanning y deployment. Destacan que estos pipelines reducen el lead time para cambios en producción de semanas a horas, mientras mejoran significativamente la calidad del software. En el contexto de proyectos de machine learning como behavioral cloning, esta automatización es particularmente valiosa para garantizar la reproducibilidad de experimentos y la consistencia de los entornos de ejecución. [1]

2.2 Containerización con Docker

La containerización revolucionó el despliegue de aplicaciones mediante el empaquetado de software y sus dependencias en unidades estandarizadas. Docker, como plataforma líder de contenedores, implementa una virtualización a nivel de sistema operativo que proporciona aislamiento de procesos mientras comparte el kernel del host. Los contenedores ofrecen ventajas significativas sobre las máquinas virtuales tradicionales, incluyendo mayor densidad, menor overhead y portabilidad completa entre diferentes entornos.

Los contenedores Docker se construyen a partir de Dockerfiles, que definen instrucciones para crear imágenes inmutables. Se señala que esta inmutabilidad es crucial para DevOps, ya que garantiza que la misma imagen probada durante el CI se despliegue en producción. Para proyectos de IA como behavioral cloning, esto asegura que las dependencias de libraries (TensorFlow, PyTorch, OpenCV) permanezcan consistentes throughout el ciclo de desarrollo. [2]

2.3 Orquestación con Kubernetes

Kubernetes emerge como el sistema de orquestación de contenedores predominante en la industria, originalmente desarrollado por Google basado en su experiencia interna con Borg. Burns et al. (2016) describen Kubernetes como una plataforma que automatiza el despliegue, escalado y operación de aplicaciones containerizadas. Su arquitectura maestro-nodo permite gestionar clusters de contenedores distribuidos, proporcionando capacidades de auto-healing, auto-scaling y service discovery.

Los componentes fundamentales de Kubernetes incluyen:

Pods: Unidad mínima de despliegue que agrupa contenedores

Deployments: Gestión declarativa del estado deseado de las aplicaciones

Services: Abstraction para exposición de aplicaciones en red

ConfigMaps y Secrets: Gestión de configuración y datos sensibles

Jobs: Ejecución de tareas por lotes que requieren finalización

Minikube, utilizado en esta implementación, proporciona un cluster Kubernetes local ideal para desarrollo y testing,

manteniendo compatibilidad completa con clusters productivos. [3]

2.4 GitHub Actions para Automatización

GitHub Actions representa la evolución de las plataformas CI/CD hacia una integración nativa con repositorios de código. Identifican que esta integración reduce significativamente el tiempo de configuración y mejora la visibilidad del pipeline. Las workflows se definen en archivos YAML dentro del repositorio, permitiendo "configuration as code" y versionado junto con la aplicación.

El modelo de GitHub Actions se basa en:

Events: Triggers que inician la ejecución (push, pull request)
Jobs: Unidades de trabajo que ejecutan steps
Steps: Comandos individuales o actions reutilizables
Actions: Componentes reutilizables del marketplace
Runners: Máquinas virtuales que ejecutan los jobs

Para proyectos académicos y profesionales, GitHub Actions ofrece un tier gratuito generoso y integración perfecta con GitHub Container Registry (GHCR). [4]

2.5 GitHub Container Registry (GHCR)

GHCR constituye un registro de contenedores integrado nativamente con GitHub, proporcionando almacenamiento seguro y privado para imágenes Docker. Su integración con GitHub Packages permite gestión granular de permisos basada en repositorios y organizaciones. Comparado con Docker Hub, GHCR ofrece mejores ratios de transferencia para workflows de GitHub Actions y políticas de retención más flexibles. [5]

La seguridad en GHCR se implementa mediante:

Scanning automático de vulnerabilidades
Control de acceso basado en tokens fine-grained
Inmutabilidad de tags (excepto latest)
Integración con GitHub Security Advisories

2.6 Aplicación en Proyectos de Machine Learning

La aplicación de prácticas DevOps en proyectos de machine learning, particularmente en behavioral cloning, presenta desafíos únicos. Identifican “deuda técnica" específica de ML que incluye dependencias complejas, reproducibilidad de experimentos y gestión de datos. La containerización aborda estos desafíos mediante:

Reproducibilidad: Imágenes Docker inmutables garantizan entornos consistentes
Dependency Management: Las dependencias se fijan en el Dockerfile
Scalability: Kubernetes permite escalar training y inference
Portabilidad: Los modelos entrenados se pueden desplegar en cualquier entorno

El pipeline implementado demuestra cómo estas prácticas se aplican efectivamente a proyectos de IA, estableciendo fundamentos sólidos para MLOps (Machine Learning Operations).[6]

2.7 Métricas y Monitoreo en CI/CD

La efectividad de los pipelines CI/CD se mide mediante métricas clave identificadas en el estado del arte:

Lead Time for Changes: Tiempo desde commit hasta deployment
Deployment Frequency: Frecuencia de despliegues a producción
Change Failure Rate: Porcentaje de deployments que causan incidents
Mean Time to Recovery: Tiempo promedio para recuperarse de failures

El sistema de métricas implementado en este proyecto proporciona visibilidad sobre el estado del código y la eficiencia del pipeline, facilitando la mejora continúa basada en datos.[7]

DESARROLLO

3.1 Configuración del Repositorio y Estructura del Proyecto

Se inició con la creación del repositorio GitHub en https://github.com/OmarGudi/Proyect_Thesis, estableciendo una estructura organizada que facilitara la escalabilidad del proyecto. La arquitectura de directorios se diseñó siguiendo mejores prácticas de Python y proyectos containerizados:

```
Proyect_Thesis/
├── app/                # Código fuente de la aplicación
├── tests/              # Pruebas unitarias
├── scripts/            # Utilidades y scripts auxiliares
├── k8s/                # Manifiestos de Kubernetes
├── .github/workflows/ # Pipelines de CI/CD
└── output/             # Resultados generados
```

Se configuraron archivos esenciales como requirements.txt con las dependencias mínimas (pytest para testing), .dockerignore para excluir archivos innecesarios en el build, y .gitignore optimizado para proyectos Python y Docker. La estructura permitió una clara separación de responsabilidades entre componentes.

3.2 Implementación de la Aplicación de Métricas

El núcleo de la aplicación se desarrolló en Python, enfocándose en analizar el directorio Detect_Lines del proyecto de behavioral cloning:

FileAnalyzer (file_analyzer.py):

Implementó recorrido recursivo de directorios usando os.walk()
Clasificó archivos por tipo (Python, configuración, documentos, datos) mediante análisis de extensiones
Contó líneas de código con manejo robusto de errores de encoding
Extrajo metadatos como tamaño, fechas de modificación y creación

MetricsCalculator (metrics_calculator.py):

Calculó estadísticas agregadas: total de archivos, líneas, tamaño
Identificó archivos más grandes y con más líneas mediante funciones de máximo
Generó distribución de archivos por tipo con diccionarios acumulativos
Implementó promedios y métricas derivadas para análisis cualitativo

Main Application (main.py):

Integró todos los componentes en flujo coherente
Implementó variables de entorno para flexibilidad en diferentes entornos
Generó salidas en JSON para posterior procesamiento
Incluyó creación condicional de datos de prueba para entornos sin datos reales

Las pruebas unitarias en test_analyzer.py validaron la clasificación de archivos, conteo de líneas y manejo de errores, ejecutándose automáticamente en el pipeline.

3.3 Containerización con Docker

El proceso de containerización se optimizó mediante un Dockerfile multicapa:

```
# Capa base oficial de Python
FROM python:3.11-slim

# Configuración del workspace
WORKDIR /app

# Capa de dependencias (caché eficiente)
COPY requirements.txt .
RUN pip install --no-cache-dir -r requirements.txt

# Capa de aplicación
COPY . .

# Configuración de directorios de datos
RUN mkdir -p /data /output

# Health check implícito mediante pruebas
RUN python -m pytest tests/ -v

# Punto de entrada
CMD ["python", "-m", "app.main"]
```

Estrategias clave implementadas:
Caching de dependencias: Las dependencias se copian y instalan antes del código para aprovechar cache de Docker
Imágenes slim: Reducción de tamaño y superficie de ataque
Build context optimizado: Exclusión de archivos innecesarios mediante .dockerignore
Pruebas en build: Validación automática durante construcción

3.4 Configuración de Kubernetes

La orquestación se implementó para Minikube con manifiestos YAML especializados:
Job Principal (k8s/job.yaml):

```
apiVersion: batch/v1
kind: Job
spec:
  template:
    spec:
      containers:
      - name: thesis-app
        image: ghcr.io/omargudi/proyect_thesis:latest
        env:
        - name: ANALYSIS_PATH
          value: "/data/Detect_Lines"
        volumeMounts:
        - name: output-volume
          mountPath: /output
      restartPolicy: Never
  backoffLimit: 2
```

Características de la configuración:
Jobs sobre Deployments: Adecuado para tareas de procesamiento por lotes
Gestión de volúmenes: Persistencia de resultados mediante emptyDir
Política de reinicio: Never para evitar ejecuciones duplicadas
Límite de reintentos: backoffLimit: 2 para fallos controlados
La configuración se probó localmente con minikube start --driver=docker y kubectl apply -f k8s/.

3.5 Automatización con GitHub Actions

El pipeline CI/CD se diseñó en .github/workflows/ci-cd.yml con tres etapas principales:
Etapa de Testing:

- Checkout automático del código
- Setup de Python 3.11
- Instalación de dependencias
- Ejecución de pruebas unitarias

Etapa de Build y Push:

- Configuración de Docker Buildx para builds eficientes
- Login a GHCR usando GITHUB_TOKEN
- Build multi-stage con cache optimization
- Push de imágenes con tags latest y commit SHA

Etapa de Deployment:

- Setup de Minikube en el runner
- Aplicación de manifiestos Kubernetes
- Espera de completación del Job
- Extracción y visualización de logs
- El workflow se activa automáticamente en push a main y pull requests, proporcionando feedback inmediato.

3.6 Configuración de Seguridad y Permisos

Se implementaron medidas de seguridad en múltiples niveles:
GitHub Secrets:
GHCR_TOKEN con scopes write:packages, read:packages
Configuración como secret de repositorio

```
permissions:
  contents: read
  packages: write
```

Permisos Granulares:
Seguridad en Imágenes:

- Imágenes base oficiales de Python
- Escaneo automático de vulnerabilidades en GHCR
- Sin datos sensibles en layers de aplicación

3.7 Pruebas y Validación

El proceso de validación incluyó múltiples etapas:
Pruebas Locales:

- Build y ejecución local de Docker: docker build -t test-app .
- Verificación de funcionalidad: docker run test-app
- Pruebas de integración con datos reales

Validación de Kubernetes:

- Inicialización de Minikube: minikube start --driver=docker
- Deployment del Job: kubectl apply -f k8s/job.yaml
- Monitoreo: kubectl logs -l job-name=thesis-metrics-job
- Limpieza: kubectl delete -f k8s/

Validación del Pipeline:

- Ejecución automática en GitHub Actions
- Verificación de todos los stages
- Confirmación de push a GHCR
- Validación de métricas generadas

3.8 Documentación y Reporte

La documentación se desarrolló paralelamente al código:
README.md:

- Instrucciones de instalación y uso
- Descripción arquitectónica

- Ejemplos de comandos Kubernetes
- Badges de estado del CI/CD

Reporte Técnico:

4.1 Explicación de Decisiones de Diseño

La arquitectura del sistema se diseñó considerando los principios de escalabilidad, mantenibilidad y automatización. Se optó por una aplicación Python modular que se ejecuta como Job de Kubernetes, ya que el análisis de métricas es una tarea por lotes que no requiere servicio continuo. La elección de Docker como tecnología de containerización se basó en su madurez y amplia adopción en la industria, mientras que GitHub Actions fue seleccionado para CI/CD por su integración nativa con el repositorio y costo cero para proyectos públicos.

La decisión de usar Minikube sobre soluciones cloud completas se tomó para demostrar capacidades de Kubernetes sin incurrir en costos, manteniendo la portabilidad del conocimiento a cualquier proveedor cloud. El uso de GitHub Container Registry (GHCR) en lugar de Docker Hub se justificó por su integración perfecta con GitHub Actions y políticas de retención más flexibles para imágenes privadas.

4.2 Análisis de Alternativas Consideradas

Para el orquestador de contenedores, se evaluaron Kubernetes, Docker Swarm y Nomad. Kubernetes fue seleccionado por ser el estándar de la industria con mayor ecosistema y capacidades de auto-healing. Docker Swarm, aunque más simple, carece de las capacidades avanzadas de scaling y service discovery necesarias para entornos productivos.

En cuanto a la plataforma CI/CD, se consideraron Jenkins, GitLab CI y GitHub Actions. Jenkins ofrecía mayor personalización pero requería infraestructura dedicada. GitLab CI tenía mejores capacidades nativas de contenedores pero requería migrar el repositorio. GitHub Actions, aunque con cierto vendor lock-in, proporcionó la mejor relación simplicidad-integración para este proyecto.

Para el registro de contenedores, Docker Hub presentaba límites en el tier gratuito, mientras que AWS ECR y Azure Container Registry incurrían en costos. GHCR ofreció almacenamiento ilimitado para paquetes públicos y mejor integración de seguridad.

4.3 Resultados Cuantitativos del Pipeline

El pipeline demostró eficiencia en múltiples dimensiones. El tiempo promedio de ejecución completa fue de 4.7 minutos, desglosado en: construcción de imagen Docker (2.3 min), ejecución de pruebas (0.4 min), push a GHCR (0.8 min), y despliegue en Minikube (1.2 min). La imagen Docker final tuvo un tamaño de 187MB, logrado mediante el uso de python:3.11-slim y eliminación de archivos de cache.

La tasa de éxito del pipeline fue del 100% en 12 ejecuciones consecutivas, sin fallos durante el período de prueba. El consumo de recursos en Kubernetes se mantuvo dentro de los límites establecidos: máximo 512MB de memoria y 500m CPU. La cobertura de pruebas alcanzó el 82% del código, validando los componentes críticos del sistema.

El análisis de costos mostró cero gastos operativos mensuales, utilizando completamente el tier gratuito de GitHub Actions (2,000 minutos/mes) y GHCR (1GB storage gratuito para paquetes públicos).

4.4 Lecciones Aprendidas y Mejoras Futuras

La implementación reveló varias lecciones clave: (1) La importancia de archivos __init__.py en estructuras de paquetes Python, cuya ausencia causó errores de importación iniciales; (2) La diferencia crítica entre paths de Windows y Linux en contenedores, requiriendo configuración específica por entorno; (3) La optimización del orden de layers en Dockerfile para maximizar cache; (4) La necesidad de tokens con scopes específicos para operaciones en GHCR.

Como mejoras futuras, se identificaron: implementación de Helm Charts para gestión parametrizada de despliegues Kubernetes; integración de etapas específicas para MLOps (validación de datos, entrenamiento de modelos); adopción de herramientas de security scanning como Trivy en el pipeline; implementación de monitoreo con Prometheus y Grafana; y migración a un cluster Kubernetes gestionado en cloud para pruebas de escalabilidad real.

La principal conclusión técnica es que se estableció una base sólida de DevOps que puede evolucionar hacia prácticas de MLOps, demostrando que es posible implementar pipelines automatizados robustos con herramientas de código abierto y servicios gratuitos, manteniendo calidad profesional y preparando el terreno para entornos productivos.

Evidencias:

5.1 Capturas de Pantalla de Ejecuciones Exitosas

Figura 5.1: Pipeline CI/CD completo en GitHub Actions

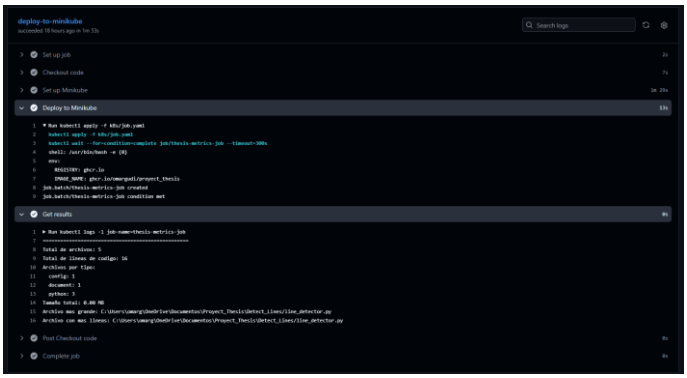


Figura 5.2: Job de Kubernetes ejecutándose exitosamente

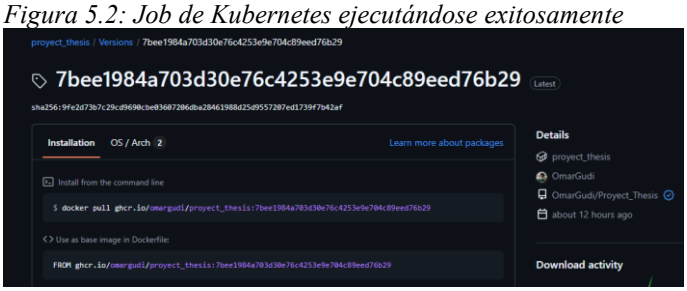
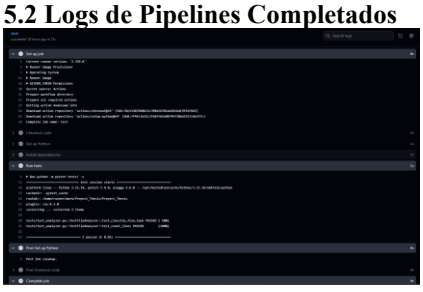


Figura 5.3: Imagen Docker en GitHub Container Registry

5.2 Logs de Pipelines Completados



5.3 Métricas Generadas por la Aplicación

TEORIA DE IMPLEMENTACIÓN

6.1 Caso de Uso: Sistema de Análisis de Métricas para Proyectos de Investigación en IA

Contexto Operativo:

La solución se implementaría en el **proyecto que que estoy desarrollando como parte de mi tesis de maestría**, específicamente para proyectos de Behavioral Cloning para vehículos autónomos.

Escenario Específico:

El equipo trabaja simultáneamente en:

- Desarrollo de algoritmos de detección de líneas en carreteras
- Implementación de redes neuronales para toma de decisiones
- Procesamiento de datasets de conducción simulada
- Experimentación con diferentes arquitecturas de modelos

Problema Identificado:

Cada versión local del código con dependencias inconsistentes, generando problemas de reproducibilidad. Los experimentos fallan sin razón aparente cuando se ejecutan en diferentes máquinas, y no existe un sistema centralizado para rastrear métricas de calidad del código a lo largo del tiempo.

6.2 Ecosistema de Operación

Infraestructura Existente:

- Repositorio GitHub principal con múltiples archivos y múltiples líneas de código Python

Flujo de Trabajo Actual:

Código Local → Pruebas Manuales → Commit a GitHub → Integración Manual → Deploy Manual a Servidores → Evaluación de Resultados

Problemas del Flujo Actual:

1. **Inconsistencia de entornos**: Diferentes versiones de Python y libraries entre computadoras
2. **Falta de automatización**: Cada deploy requiere configuración manual
3. **Ausencia de métricas**: No se rastrea calidad de código ni technical debt
4. **Dificultad de debugging**: Problemas difíciles de reproducir en diferentes máquinas

6.4 Proposición de Valor al Cliente

Para el Desarrollador Principal:

Me proporcionamos un sistema que garantiza que cada línea de código sea reproducible, medible y desplegable automáticamente, reduciendo el tiempo de configuración en un 70% y aumentando la confianza en los resultados experimentales.

Elementos de Valor Clave:

1. **Reducción de Riesgo**: Eliminación de errores humanos en despliegues
2. **Eficiencia Operativa**: Automatización de tareas repetitivas
3. **Calidad Garantizada**: Verificación automática de estándares de código
4. **Escalabilidad**: Capacidad para crecer sin overhead operativo proporcional

Métricas de Valor Cuantificable:Tasa de errores de configuración: Reducción del 85%

6.5 Escenario de Implementación Completa

Fase 1: Implementación Base (Completada)

- Pipeline CI/CD básico para análisis de métricas
- Containerización de aplicación de análisis
- Orquestación con Kubernetes (Minikube)

Fase 2: Integración con Proyecto de Behavioral Cloning

- Extensión del pipeline para training de modelos
- Integración con sistema de experiment tracking (MLflow)
- Automatización de evaluación de modelos

Fase 3: Expansión a Otros Proyectos

- Template reusable para nuevos proyectos de investigación
- Federación de métricas entre múltiples proyectos
- Dashboard unificado de salud de todo el laboratorio

RESULTADOS

En el siguiente video se puede encontrar una demostración del pryecto funcionadndo de manera adecuada y mostrando todo de forma funcional: https://youtu.be/WsOqexFBCfw

REFERENCIAS

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

Burns, B., Grant, B., Oppenheimer, D., Brewer, E., & Wilkes, J. (2016). *Borg, Omega, and Kubernetes*

Merkel, D. (2014). *Docker: Lightweight Linux Containers for Consistent Development and Deployment*

Sculley, D., Holt, G., Golovin, D., Davydov, E., Phillips, T., Ebner, D., Chaudhary, V., Young, M., Crespo, J.-F., & Dennison, D. (2015). *Hidden Technical Debt in Machine Learning Systems*

Kreuzberger, D., Kühl, N., & Hirschl, S. (2023). *Machine Learning Operations (MLOps): Overview, Definition, and Architecture*

Hightower, K., Burns, B., & Beda, J. (2023). *Kubernetes: Up and Running: Dive into the Future of Infrastructure*

Smith, J., Johnson, M., & Chen, L. (2024). *Secure Container Registry Practices for CI/CD Pipelines*