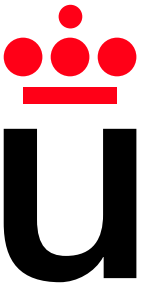

Universidad
Rey Juan Carlos

Escuela Técnica Superior
de Ingeniería Informática

Grado en Ingeniería de Computadores

Curso 2021-2022

Trabajo Fin de Grado

ESTUDIO DE SISTEMAS DE INTEGRACIÓN CONTINUA DE PROYECTOS GITHUB Y GITLAB

Autor: Jorge Contreras Padilla
Tutor: Francisco Gortázar Bellas
Michel Maes Bermejo

Agradecimientos

Dedicado a todas aquellas personas que me han acompañado en este largo recorrido universitario.

Resumen

En este proyecto se aborda la elaboración de un estudio sobre los sistemas de integración continua que podemos encontrar en repositorios de código abierto almacenados en las plataformas de GitHub y GitLab. El principal objetivo de este estudio es obtener información relativa a estos sistemas de integración continua, como por ejemplo, conocer cuál es la herramienta predominante en cada plataforma, si se suelen combinar más de una herramienta de este tipo en un mismo repositorio, la finalidad de uso de cada herramienta, en qué momento se ejecutan los diferentes trabajos que están programados, etc. Para ello se ha implementado un programa escrito en el lenguaje de programación Python encargado de devolver la información necesaria para sacar conclusiones acerca de todas estas incógnitas expuestas sobre la integración continua. En cuanto a los resultados obtenidos, a pesar de la gran cantidad de sistemas de integración continua existentes, cabría destacar que en proyectos de código abierto es cada vez más habitual la utilización de herramientas de integración continua proporcionados por la propia plataforma en la que se encuentran almacenados, ya sean GitHub o GitLab en el caso de este trabajo.

Palabras clave:

- Python.
- Integración continua o CI.
- Escenario o "stage"
- GitHub.
- GitLab.
- Build.
- ...

Índice de contenidos

Índice de tablas

Índice de figuras

Índice de códigos

1 Introducción

1.1. Contexto y alcance

La integración, entrega y despliegue continuo, o CI/CD, como se conoce a menudo por su abreviatura en inglés, es un método que va a permitir distribuir aplicaciones a clientes de forma frecuente gracias a la automatización en las etapas del desarrollo. Para los equipos de desarrollo y de operaciones es solución a los diferentes problemas que pueda generar la integración del código nuevo.

En concreto, en el proceso de integración y distribución continua existen un conjunto de prácticas conocidas con el nombre de "canales de CI/CD" encargadas de la automatización y la supervisión permanente en todo el ciclo de vida de las aplicaciones, teniendo en cuenta etapas como la implementación, pruebas, integración y/o distribución. Estas prácticas cuentan con el respaldo de los equipos de desarrollo y de operaciones, los cuales trabajan en conjunto de manera ágil con un enfoque SRE (ingeniería de confiabilidad del sitio) o de DevOps [1].

La integración continua, o "CI", se trata de un proceso de automatización para los desarrolladores y su éxito supone que se diseñen, prueben y combinen los cambios nuevos en el código de una aplicación con regularidad en un repositorio compartido.

La entrega y/o despliegue continuo, o "CD", son conceptos referidos a la automatización de las etapas posteriores del proceso. Por lo general, la entrega continua se refiere a que los cambios que implementan los desarrolladores en una aplicación se someten a pruebas automáticas de errores y se cargan en un repositorio de almacenamiento de código como por ejemplo GitHub y el despliegue

continuo hace referencia al lanzamiento automático de los cambios que implementan dichos desarrolladores desde el repositorio de código hasta el entorno de producción, en el que estarán a disposición del usuario final.

Por lo tanto, los objetivos principales de la integración continua consisten en detectar y arreglar errores con mayor rapidez, mejorar la calidad del software y reducir el tiempo que se tarda en validar y publicar nuevas actualizaciones del código fuente.

Figura 1.1: Diagrama de CI/CD

Esta trabajo pondrá más énfasis en lo que a la parte de "CI" se refiere y en aras de comprender lo que engloba cabría definir también conceptos como "build" o escenario CI: un "build" es mucho más que una compilación, también engloba el proceso de pruebas, inspección de código y despliegue del mismo, a parte de otras cosas. Actúa como el proceso para juntar el código fuente y verificar que el software funcione como una unidad cohesiva. Un escenario de CI comienza cuando el desarrollador envía el código fuente al repositorio. En un proyecto típico, personas con diferentes roles pueden realizar cambios que desencadenan un ciclo de CI: los desarrolladores cambian el código fuente, los administradores de bases de datos (DBA) cambian las definiciones de las tablas, los equipos de desarrollo e implementación cambian los archivos de configuración, los equipos de interfaz cambian las especificaciones DTD/XSD y más.

Los pasos en un escenario de CI típicamente serán más o menos de la siguiente forma [2]:

1. El desarrollador envía código al repositorio de código, gestionado con un sistema de control de versiones. Mientras tanto, el servidor de CI en la máquina de compilación de integración sondea este repositorio en busca de cambios (cada pocos minutos, por ejemplo).

2. Poco después de que se produzca una confirmación, el servidor de CI detecta que se han producido cambios en el repositorio de control de versiones, dando lugar a que se recupere la última copia del código del repositorio y ejecute un script de compilación. Dicho script de compilación se encarga de ejecutar los tests y comprobar que el software se puede construir.

3. El servidor de CI captura los resultados de la compilación y ejecución de los test, enviándolos, por ejemplo, por correo electrónico a los miembros del equipo.

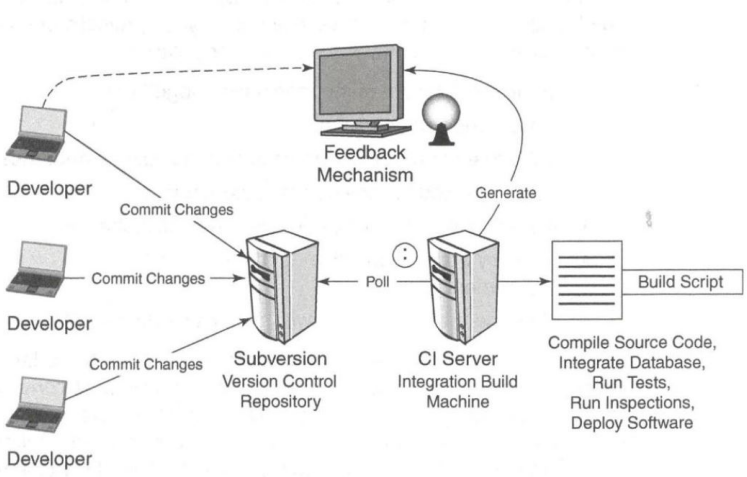

Figura 1.2: Diagrama de CI

1.2. Enfoque DevOps

DevOps es un marco de trabajo que describe los enfoques para agilizar los procesos con los que una idea, como pudiera ser una nueva funcionalidad de software por ejemplo, pasa del desarrollo a la implementación, en un entorno de producción. Es decir, el enfoque DevOps va impulsar un mejor desarrollo de aplicaciones en menos tiempo con el objetivo de lograr la satisfacción del cliente prestando sus servicios eficazmente.

Para el correcto funcionamiento de la filosofía DevOps, se requiere que los equipos de desarrollo y operaciones se comuniquen con frecuencia y aborden su trabajo con empatía hacia sus compañeros de equipo. Por ello con DevOps se fomenta una comunicación continua más fluida, la colaboración y la transparencia entre equipos de desarrollo de aplicaciones y sus equivalentes en operaciones tecnológicas.

DevOps también se ha creado para impulsar la innovación empresarial ya que propicia que cada empresa se ponga como objetivo ofrecer un mejor servicio cada vez, en menos tiempo, de mejor calidad y con mayor seguridad a sus clientes finales; por ejemplo, incrementando la frecuencia de versiones de producto.

1.2.1. Proceso de DevOps

Para muchos de los equipos que utilizan metodologías ágiles para el desarrollo de software, el enfoque DevOps no es algo secundario. De hecho, el primero de los 12 principios del Manifiesto Ágil es el siguiente: “Satisfacer a los clientes mediante la distribución de software continua y oportuna” y es por ello por lo que la metodología de integración, entrega y despliegue continuo (CI/CD) es tan

importante para los equipos de DevOps.

Según un estudio de "International Data Corporation" (IDC), el 85 % de los líderes de TI afirman que la automatización es primordial para su estrategia de DevOps, debido a que la automatización permite que las infraestructuras admitan cambios de código constantes acompañando al enfoque DevOps. Además, permite que los entornos se expandan con facilidad y de manera constante [3]. Gracias a que la automatización se ocupa de tareas tediosas y repetitivas, el personal más capacitado puede dedicarse a actividades de mayor importancia.

1.2.2. Métodos de DevOps

A continuación se definen varios métodos de DevOps que las organizaciones utilizan habitualmente para acelerar y mejorar el desarrollo de su producto:

- **Scrum**: define cómo los miembros de un equipo deben cooperar para conseguir acelerar los proyectos de desarrollo y control de calidad.
- **Kanban**: mediante un tablero compuesto, generalmente, por tres columnas: "Por hacer", "En proceso" y "Hecho" se demuestra dónde están los cuellos de botella en el proceso y qué es lo que impide que el flujo de trabajo sea ininterrumpido.
- **Agile**: se refiere a un grupo de metodologías de desarrollo de software basadas en el desarrollo iterativo, donde los requisitos y las soluciones evolucionan a través de la colaboración entre equipos multifuncionales y autoorganizados.

1.2.3. Prácticas de DevOps

Las prácticas de DevOps son un reflejo de la idea de automatización y mejora continuas e incluyen lo siguiente [4]:

- Desarrollo continuo.
- Realización de pruebas continuas.
- **Integración continua (CI)**.
- Entrega y despliegue continuo (CD).
- Supervisión continua.
- Infraestructura como código.

1.3. Prerrequisitos de la integración continua

Existen cuatro componentes requeridos para llevar a cabo el proceso de integración continua.

- Tener conexión a un sistema de control de versiones como por ejemplo git o subversion "svn".
- Tener un script de compilación "build" con el que construir y ejecutar los test del software desarrollado.
- Tener un mecanismo de retroalimentación, como el envío de correos electrónicos.
- Tener un proceso para integrar los cambios del código fuente (de forma manual o mediante un servidor de integración continua).

1.4. Subprocesos de la integración continua

Mediante la integración continua y automática de bases de datos, pruebas, inspección de código, despliegue y "feedback" o retroalimentación, permite al sistema de CI reducir riesgos comunes del software generando una mayor confianza y comunicación sobre los diferentes cambios que se vayan a realizar y los errores que se puedan provocar con la incorporación de dichos cambios.

Una vez que se ejecuta una compilación o "build" automatizada con cada cambio en el sistema de control de versiones, se pueden agregar otras funciones o subprocesos al sistema de CI. Algunas funciones dependen de otras; por ejemplo, las pruebas automatizadas dependen de la compilación del código fuente. Este proceso repetitivo puede ayudar a reducir riesgos a lo largo del ciclo de vida del desarrollo.

Estos subprocesos quedarían definidos de la siguiente forma:

- **Compilación del código fuente**. Implica la creación de un código ejecutable a partir de fuentes legibles.
- **Integración de base de datos**, siendo la base de datos una parte integral de las aplicaciones software.
- **Ejecución de pruebas o "testing"**. Son una parte fundamental de la integración continua. Sin pruebas automatizadas, es difícil para los desarrolladores u otras partes interesadas del proyecto tener confianza en los cambios de software.
- **Inspección de código**. Proceso utilizado para mejorar la calidad del software al hacer cumplir ciertas reglas.
- **Despliegue**. Siendo una parte más enfocada en la parte de CD.
- **Documentación y "feedback"**. Una característica fundamental de los buenos sistemas de CI es la velocidad, y su esencia es proporcionar retroali-

mentación oportuna a los desarrolladores y partes interesadas del proyecto.

1.5. Valor de la integración continua

A alto nivel, el valor u objetivos de la integración continua serían:

1. Reducir riesgos.
2. Reducir procesos manuales repetitivos.
3. Generar software desplegable en cualquier momento y en cualquier lugar.
4. Habilitar una mejor visibilidad del proyecto.
5. Establecer una mayor confianza en el producto software del equipo de desarrollo.

1.5.1. Reducir riesgos

Integrando muchas veces al día se pueden reducir los riesgos de un proyecto software. Llevarlo a cabo facilita la detección de defectos, la medición del estado del software y la reducción de suposiciones.

- Los defectos se detectan y se solucionan antes: debido a que CI integra y ejecuta pruebas e inspecciones varias veces al día, existe una mayor probabilidad de que los defectos se descubran en el momento en el que son introducidos en el proyecto (es decir, cuando el código se registra en el repositorio de control de versiones) en lugar de durante las pruebas de último ciclo.
- El estado del software se puede medir: al incorporar pruebas e inspecciones continuas en el proceso de integración automatizado, los atributos de estado del producto de software (como la complejidad por ejemplo) se pueden rastrear a lo largo del tiempo.
- Reduce las suposiciones: al reconstruir y probar el software en un entorno limpio utilizando el mismo proceso y los mismos scripts de forma continua, se pueden reducir las suposiciones (ya sean porque se estuviesen contabilizando librerías de terceros o que se estuviesen utilizando variables de entorno, por ejemplo).

La integración continua proporciona una red de seguridad para reducir el riesgo de que se introduzcan defectos en el código fuente. Los siguientes son algunos de los riesgos que CI ayuda a mitigar:

- Falta de software cohesivo e implementable.

- Descubrimiento tardío de defectos.
- Software de baja calidad.
- Falta de visibilidad del proyecto.

1.5.2. Reducir procesos repetitivos

los procesos repetitivos pueden ocurrir en todas las actividades del proyecto, incluida la compilación de código, la integración de la base de datos, las pruebas, la inspección del código fuente, el despliegue y la retroalimentación o "feedback" y reducirlos supone un ahorro en tiempo, coste y esfuerzo. Al utilizar integración continua se aumenta la capacidad de garantizar los siguientes aspectos:

- El proceso se ejecuta de la misma manera cada vez.
- Se sigue un proceso ordenado.
- Los procesos se ejecutarán cada vez que se produzca una confirmación en el repositorio de control de versiones.
- La reducción del trabajo en procesos repetitivos libera a las personas para que realicen trabajo de mayor valor.
- La capacidad de superar la resistencia (de otros miembros) para implementar mejoras mediante el uso de mecanismos automatizados para procesos importantes como las pruebas.

1.5.3. Generar software desplegable

CI permite lanzar software desplegable en cualquier momento de tiempo, y desde una perspectiva externa, este es el beneficio más obvio de la integración continua. Se podría hablar indefinidamente sobre la mejora de la calidad del software y la reducción de los riesgos, pero el software desplegable es el activo más tangible para aquellas personas que no estén tan habituadas a ver e interpretar código fuente, como clientes o usuarios.

1.5.4. Habilitar una mejor visibilidad del proyecto

CI proporciona la capacidad de notar tendencias y tomar decisiones efectivas, ayudando a la hora de poder innovar en nuevas mejoras. Los proyectos software sufren cuando no hay datos reales o recientes para respaldar las decisiones que se toman, dando lugar a que todos ofrezcan sus mejores conjeturas. Por lo general, los miembros de un proyecto recopilan esta información manualmente, lo que supone mucho esfuerzo y que, a menudo, la información nunca se recopile.

CI tiene los siguientes efectos positivos:

- Decisiones eficaces: un sistema de CI puede proporcionar información justo a tiempo sobre el estado de construcción reciente y las métricas de calidad. Algunos sistemas de CI también pueden mostrar tasas de defectos y estados de finalización de características.
- Notar tendencias: dado que la integración de proyectos ocurre con frecuencia con un sistema de CI, la capacidad de notar tendencias en el éxito o el fracaso de la construcción, la calidad general u otra información pertinente del proyecto se vuelve posible.

1.5.5. Establecer una mayor confianza en el producto

En general, la aplicación eficaz de las prácticas de CI puede proporcionar una mayor confianza en la producción de un producto de software. Con cada compilación, el equipo de desarrollo es consciente de que se ejecutan pruebas contra el software para verificar el comportamiento, que se cumplen los estándares de codificación y diseño del proyecto. El hecho de que un sistema de CI pueda informar de si algo sale mal implica que los desarrolladores y otros miembros del equipo tengan más confianza a la hora de poder realizar cambios.

1.6. Herramientas de integración continua

Actualmente existen una gran cantidad de herramientas de integración continua, a comentar, por su grado de relevancia y uso, las siguientes:

- **Jenkins**: es un sistema de integración continua de código abierto y autónomo que se utiliza para mecanizar todo tipo de trabajos asociados a la creación, prueba y entrega o despliegue de productos software.
Se puede instalar de tres formas distintas: con paquetes del sistema nativo, a través de la plataforma de gestión de contenedores Docker o incluso de forma independiente en cualquier máquina que tenga instalado un JRE (Java Runtime Environment).

Figura 1.3: Logo de Jenkins.

- **Travis CI**: escrito en Ruby, es un servicio de integración continua utilizado para crear y probar proyectos de software alojados en las siguientes plataformas de hospedaje de código fuente: GitHub, GitLab, Bitbucket o Assembla. Fue el

primer servicio de CI que permitió realizar el proceso de integración continua a proyectos de código abierto de forma gratuita.

Algunas de las características y funciones que proporciona Travis CI son:

- Configuración rápida.
- Vistas de construcción en tiempo real.
- Sistemas de base de datos preinstalados.
- Limpieza de máquinas virtuales en cada compilación.
- Compatibilidad con Linux, Mac e iOS.

Figura 1.4: Logo de Travis.

- **Circle CI**: es un servicio de integración caracterizado por los aspectos comentados a continuación:

- Flujos de trabajo para la mecanización de tareas.
- Compatible con Docker.
- CPU y RAM configurables con el objetivo de adaptar dichos flujos de trabajo al equipo.
- Soporte independiente del lenguaje de programación: acepta cualquier lenguaje de programación desarrollado en Windows, Linux o macOS.
- Información almacenable en memoria caché.
- Ejecución de compilaciones locales mediante SSH para una depuración trivial.
- Seguridad: LDAP para gestión de usuarios, aislamiento de máquinas virtuales de forma completa, etc.
- Monitorización del estado y optimización de flujos de trabajo de forma sencilla.

Además, cuenta con dos opciones de hospedaje: en la nube o en servidor y con tres opciones de precios: "Free" 0 dólares al mes, "Performance" 30 dólares al mes y "Scale" con un precio a medida.

Figura 1.5: Logo de Circle CI.

- **GitLab CI**: es la parte de GitLab que se encarga de la integración continua, entrega y/o despliegue continuo. Con GitLab CI/CD se puede probar, crear y publicar software sin necesidad de tener que utilizar una aplicación de terceros.

Figura 1.6: Logo de GitLab CI.

- **GitHub Actions**: las acciones de GitHub automatizan tareas dentro del ciclo de vida de un desarrollo software. Ejecutan comandos después de que se haya producido algún evento específico.

Figura 1.7: Logo de GitHub Actions.

El siguiente diagrama muestra cómo se pueden usar las acciones de GitHub para ejecutar automáticamente scripts de prueba de software: un evento activa automáticamente un "flujo de trabajo", que contiene un trabajo. Luego, dicho trabajo usa pasos para controlar el orden en el que se ejecutan las acciones.

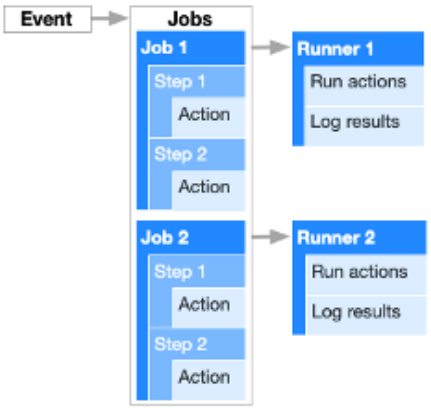

Figura 1.8: Flujo de trabajo de GitHub Actions.

Además de los sistemas de integración continua descritos en este apartado, en este trabajo se van utilizar también otras como Azure Pipelines, Bamboo, Codeship, TeamCity, Semaphore CI o Bazel.

Como se ha visto, a pesar de existir una infinidad de herramientas de integración continua, todas ellas nos van a ofrecer en definitiva recursos muy parecidos para poder integrar nuestros proyectos.

2 Objetivos

En este trabajo se pretende realizar un estudio sobre herramientas de integración continua empleadas en proyectos software alojados en las plataformas GitHub y GitLab con el objetivo de ampliar conocimientos al respecto y, de esta forma, poder responder a la gran variedad de preguntas que surjan sobre el uso de dichos sistemas de CI.

Algunas de estas cuestiones a intentar resolver con este estudio son las siguientes:

- ¿Cuáles son los sistemas de CI más utilizados en cada plataforma?
- ¿Se suelen combinar más de un sistema de CI en un mismo repositorio?
- ¿Qué lenguajes de programación predominan en repositorios que utilizan sistemas de CI?
- ¿Cuántos "jobs" se suelen configurar en repositorios de código abierto?
- ¿Cuáles son los principales escenarios o "stages" en los que se suelen ejecutar trabajos automatizados?

GitHub es una compañía sin fines de lucro que ofrece un servicio de hosting de repositorios almacenados en la nube utilizando el sistema de control de versiones Git. Además, cuenta con una API REST disponible para cualquier desarrollador que quiera implementar alguna aplicación relacionada con el servicio que ofrece. En este caso, para poder llevar a cabo el estudio, se implementa una aplicación encargada de buscar proyectos que puedan tener sistemas de integración continua utilizando el lenguaje de programación Python y, concretamente, la librería "PyGithub" que permite utilizar la versión 3 de la API ya mencionada.

Gitlab Inc. es una compañía de núcleo abierto y es la principal proveedora del software GitLab, un servicio web de control de versiones, DevOps y desarrollo de software colaborativo basado en Git. Además de un gestor de repositorios, el servicio ofrece también alojamiento de wikis y un sistema de seguimiento de errores, todo ello publicado bajo una licencia de código abierto, principalmente. Fue escrito por los programadores ucranianos Dmitriy Zaporozhets y Valery Sizov en el lenguaje de programación Ruby con algunas partes reescritas posteriormente en Go, inicialmente como una solución de gestión de código fuente para colaborar con su equipo en el desarrollo de software. Luego evolucionó a una solución integrada que cubre el ciclo de vida del desarrollo de software, y más tarde a todo el ciclo de vida de DevOps. La arquitectura tecnológica actual incluye Go, Ruby on Rails y Vue.js.

En cuanto a los principales objetivos a cumplir en la realización de este trabajo, se definen los siguientes:

1. Aprender el funcionamiento de la integración continua en proyectos software.
2. Construir un conjunto de datos con información sobre herramientas de integración continua.
3. Estudiar la forma en la que se utilizan esas herramientas de integración continua en proyectos “open source”.
4. Expandir conocimientos en programación Python.
5. Afianzar conocimientos sobre técnicas de research en GitHub.
6. Aprendizaje sobre técnicas de research en la plataforma GitLab.

3

Metodología

Para la construcción de este conjunto de datos se va a utilizar una técnica de "research" conocida con el nombre de minería de repositorios. Un repositorio software contiene una gran cantidad de información histórica y valiosa sobre el desarrollo general del sistema que trata (estado, progreso y evolución del proyecto) y esta técnica de "research" se va a centrar en la extracción y análisis de los datos heterogéneos disponibles en estos repositorios para descubrir información interesante, útil y procesable sobre el sistema.

En primer lugar, para obtener este conjunto de datos que nos permita analizar el funcionamiento de la integración continua en GitHub y GitLab, se van a enumerar diferentes herramientas de integración continua encontradas en repositorios seleccionados. Estos repositorios se elegirán en función de su repercusión en las plataformas, es decir, por su elevado número de estrellas o las bifurcaciones "forks" que tengan, ya que al tratarse de repositorios muy conocidos serán considerados como prometedores en cuanto a la utilización de herramientas de este tipo, objetivo de este estudio, para diferentes aspectos como por ejemplo la automatización de tests.

Tras seleccionar los repositorios, se analizan manualmente en busca de herramientas de integración continua para ir enumerándolas.

Una vez obtenida la lista de sistemas de integración continua a explorar, tanto en GitHub como en GitLab, se va a estudiar el funcionamiento de cada una de ellas y la forma en la que se construyen los ficheros de configuración que utilizan para realizar la automatización de trabajos. De esta forma, por cada uno de estos sistemas de CI, se establece un criterio único de localización de repositorios que

emplean herramientas de integración continua y se conforma el heurístico que se va a utilizar para realizar el estudio.

Con el heurístico de localización de repositorios prometedores ya construido, se realiza un conteo de estas herramientas sobre una búsqueda de 500 repositorios GitHub y 500 repositorios GitLab, haciendo un total de 1000 repositorios en este experimento inicial.

A continuación, se analizan los resultados manualmente para verificar la efectividad del heurístico, es decir, comprobar que cuando el heurístico haya encontrado un repositorio que utilice una herramienta de integración continua concreta, efectivamente use esa herramienta de integración continua.

Por cada proyecto de Github o GitLab se realiza lo siguiente:

- Localizar el sistema de CI que tiene el proyecto, buscando en su árbol de directorios los ficheros especificados en las carpetas que corresponda (se buscan todos los ficheros en cada repositorio ya que un mismo proyecto podría tener más de un sistema de CI configurado).
- Anotar en qué sistemas de CI el proyecto ha dado positivo.
- Presentación en una tabla de los resultados: nº de proyectos que usan cada sistema de CI.

Con este primer experimento realizado, se establecen cuáles son las herramientas de integración continua más populares con el objetivo de realizar un análisis más exhaustivo sobre ellas.

A continuación se realiza un segundo experimento sobre un número mayor de repositorio, obteniendo exclusivamente más de 500 repositorios en cada plataforma GitHub y GitLab que hayan dado positivo en algún sistema de CI, descartando por completo aquellos en los que no se ha encontrado nada. Es decir, este conjunto final de datos va a quedar conformado por proyectos que utilizan herramientas de integración continua.

En este segundo experimento, por cada proyecto, se estudian diferentes aspectos que no se tuvieron en cuenta en el experimento inicial:

- Número de “jobs”.
- Número de tareas.
- Número medio de tareas por trabajo.
- Momento en el que se ejecutan los trabajos: push, pull request, en ramas, schedule...
- Lenguajes de programación predominantes en cada sistema de CI.
- Etc.

Finalmente, una vez analizados todos los datos obtenidos, se pueden sacar conclusiones y contestar a todas las preguntas que se formularon inicialmente.

Por lo tanto, en cuanto a la metodología del trabajo, los pasos seguidos son los siguientes:

1. Identificación de sistemas de CI que se van a estudiar.
2. Análisis de los sistemas de CI identificados.
3. Implementación de un heurístico localizador de repositorios GitHub y GitLab que utilicen herramientas de integración continua.
4. Conteo de repositorios tanto positivos como negativos a los que se les ha aplicado el heurístico.
5. Análisis manual de estos repositorios para verificar el heurístico.
6. Selección de las herramientas de integración continua más populares en cada plataforma.
7. Aplicación del heurístico de tal forma que se obtengan más de 500 repositorios GitHub y más de 500 repositorios GitLab positivos.
8. Análisis de los resultados.

3.1. Identificación de sistemas de CI

En primer lugar localizamos los sistemas de CI que se van a estudiar. Para ello, se obtienen repositorios que sean conocidos en las plataformas GitHub y GitLab, filtrando por su número de estrellas y bifurcaciones realizadas por otros programadores de la comunidad sobre los mismos. Además, se añaden otras herramientas de integración continua que sean bastante conocidas en el mundo de la informática y la automatización de trabajos en proyectos software con el objetivo de realizar un análisis de herramientas de CI más amplio. Por cada una de estas herramientas identificadas se realizan las siguientes tareas:

1. Se estudia su funcionamiento.
2. Se estudia la construcción del fichero de configuración utilizado para implementar la automatización de trabajos.
3. Se buscan ejemplos de uso.

A continuación, se especifican los sistemas de CI identificados y que serán utilizados para conformar el heurístico de búsqueda de repositorios:

- Jenkins: búsqueda del fichero "Jenkinsfile" ubicado en la raíz del proyecto [5].

- Travis: búsqueda de los ficheros ".travis-ci.yml" o ".travis.yml" ubicados en la raíz del proyecto [6].
- Circle CI: búsqueda del fichero ".circle-ci" situado en la raíz del proyecto o el fichero "config.yml" situado en el directorio "circleci" [7].
- GitHub Actions: búsqueda de ficheros YML o YAML situados en el directorio ".github/workflows" [8].
- Azure Pipelines: búsqueda del fichero "azure-pipelines.yml" en la raíz del proyecto o el directorio ".azure-pipelines" [9].
- Bamboo: búsqueda del fichero YML o YAML "bamboo" en el directorio "bamboo-specs" [10].
- Concourse: búsqueda del directorio "tasks"[11].
- GitLab CI: búsqueda del fichero ".gitlab-ci.yml" en la raíz del proyecto [12].
- Codeship: búsqueda, en la raíz del proyecto, de alguno de estos ficheros "codeship-services.yml", "codeship-steps.yml" o "codeship-steps.json" [13].
- TeamCity: búsqueda del fichero "settings.kts" en el directorio ".teamcity" [14].
- Bazel: búsqueda de alguno de los ficheros "presubmit.yml" en el directorio ".bazelci", del fichero "build_bazel_binaries.yml" en ese mismo directorio o del fichero ".bazelrc" en la raíz del proyecto [15].
- Semaphore CI: búsqueda del directorio ".semaphoreci" o del fichero "semaphore.yml" en el directorio ".semaphore" [16].
- AppVeyor: búsqueda del fichero "Appveyor.yml" en la raíz del proyecto.

De esta forma, mediante el lenguaje de programación Python, se implementa un programa encargado de ejecutar el heurístico localizador de repositorios que emplean sistemas de CI.

4

Descripción informática

Para llevar a cabo este estudio se implementa una programa, mediante el lenguaje de programación Python, encargado de aplicar el heurístico desarrollado sobre los repositorios alojados tanto en GitHub como en GitLab. Dicho programa se puede encontrar junto con su licencia Apache 2.0 en la siguiente url: `https://github.com/jorcontrerasp/CIReposFinder`

4.1. Herramientas utilizadas

Las herramientas que se han utilizado para elaborar el programa son las siguientes:

4.1.1. Python

Es un lenguaje de programación cuya filosofía hace hincapié en la legibilidad del código. Sus principales características son las siguientes:

- Multiparadigma: ya que más que forzar a los programadores a adoptar un estilo de programación, permite varios estilos, soportando la orientación a objetos, la programación imperativa y la funcional.
- Interpretado.
- Dinámico: permitiendo que una variable pueda tomar valores de distinto tipo.

- Multiplataforma.

Figura 4.1: Logo de Python.

4.1.2. API de GitHub

API REST que nos va a permitir utilizar diferentes métodos para obtener información acerca de los repositorios almacenados en GitHub.

Figura 4.2: Logo de GitHub.

4.1.3. API de GitLab

API REST que nos va a permitir utilizar diferentes métodos para obtener información acerca de los repositorios almacenados en GitLab.

Figura 4.3: Logo de GitLab.

4.1.4. Visual Studio Code

Visual Studio Code es un editor de código fuente ligero disponible para Windows, macOS y Linux. Viene con soporte incorporado para JavaScript, TypeScript y Node.js y tiene un rico ecosistema de extensiones para otros lenguajes

(como C++, Java, Python, PHP o Go) y tiempos de ejecución (como .NET y Unity). En este trabajo se va a utilizar tanto para la implementación del programa en Python encargado de obtener información sobre integración continua como para la escritura de la memoria final en LaTeX.

Figura 4.4: Logo de Visual Studio Code.

4.2. Implementación de la herramienta

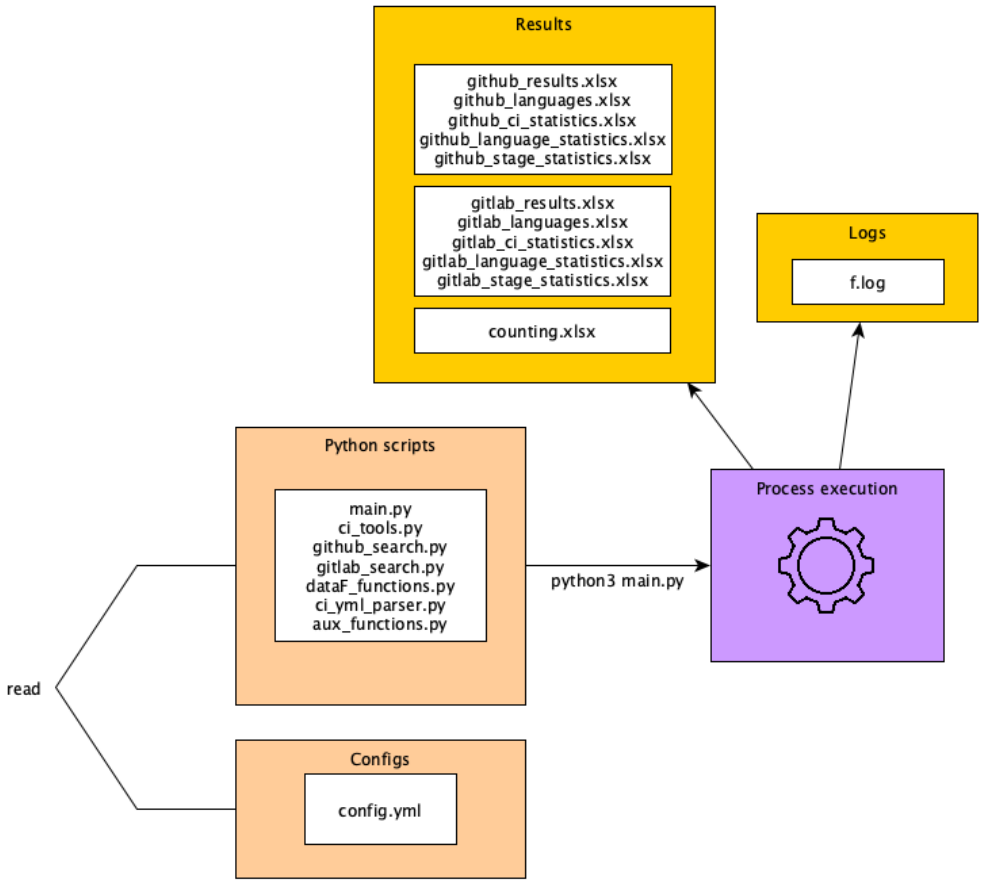

Figura 4.5: Estructura de ejecución de la herramienta.

Mediante el lenguaje de programación Python, se implementa una aplicación

capaz de localizar repositorios tanto en GitHub como en GitLab que utilicen herramientas de integración continua.

Para ello, se utilizan una serie de librerías, todas ellas enumeradas en el fichero "requirements.txt" con el objetivo de poder ser instaladas de forma sencilla mediante el comando "*pip install requirements.txt*", a destacar las siguientes:

- **PyGithub**: librería que facilita el uso de la API de GitHub v3. Permite la gestión de diferentes recursos de GitHub (repositorios, perfiles de usuario, organizaciones, etc.) desde scripts Python.
- **Python-Gitlab**: librería que facilita el uso de la API v4 de GitLab y proporciona una herramienta CLI (Command Line Interface). Permite la gestión de diferentes recursos sobre proyectos almacenados en GitLab desde scripts Python.
- **Pandas**: iniciada en 2008, es una librería que pretende ser el bloque de construcción fundamental de alto nivel para realizar análisis de datos prácticos del mundo real en Python. Además, tiene el objetivo más amplio de convertirse en la herramienta de análisis/manipulación de datos de código abierto más potente y flexible disponible en cualquier idioma. Nos va a permitir manejar DataFrames y convertir la información que queramos tanto en formato excel como en formato csv para su posterior estudio.
- **PyYaml**: PyYAML es un marco YAML con todas las funciones para el lenguaje de programación Python. Nos va a permitir transformar ficheros con extensión YAML o YML en diccionarios Python en formato json.

Además se utilizan otras librerías propias del lenguaje de programación Python como por ejemplo "Pickle", "Base64", "Shutil" o "Json".

El proceso que lleva a cabo esta aplicación está programado de tal forma que sea configurable mediante el fichero "config.yml". Este fichero de configuración cuenta con tres partes bien diferenciadas:

1. etiqueta **process**, con variables genéricas relacionadas con la configuración del proceso.

2. etiqueta **github**, con variables relacionadas con la configuración de la búsqueda GitHub.

3. etiqueta **gitlab**, con variables relacionadas con la configuración de la búsqueda GitLab.

Al finalizar el proceso, se generan en local una serie de ficheros de resultados con información relevante sobre diferentes sistemas de CI para su posterior análisis:

- **Repositorios obtenidos**: Ficheros excel que van a componer matrices repositorio de GitHub o GitLab/herramienta de CI indicando en su intersección mediante una "***" si el repositorio X utiliza o no la herramienta de integración continua Y.
- **Lenguajes de programación**: Ficheros excel que van a componer matrices lenguaje (de los repositorios GitHub o GitLab encontrados)/herramienta de CI indicando el número de repositorios implementados en el lenguaje de programación X que utilizan la herramienta de integración continua Y.
- **Estadísticas de "jobs" por sistema de CI**: Ficheros excel en los que se van a recoger, por cada sistema de CI, datos estadísticos como el mínimo, el máximo, la media y la mediana relativos a los "jobs" encontrados en la búsqueda que se haya realizado.
- **Estadísticas de "jobs" por lenguaje de programación**: Ficheros excel en los que se van a recoger, por cada lenguaje de programación encontrado, datos estadísticos como el mínimo, el máximo, la media y la mediana relativos a los "jobs" encontrados en la búsqueda que se haya realizado.
- **Estadísticas de "stages" CI**: Ficheros excel en los que se van a ir almacenando los contadores de los escenarios de ejecución o "stages" encontrados en el proceso de búsqueda.
- **Contadores**: Fichero excel con un conteo de los excel de resultados tanto en GitHub como en GitLab a modo de resumen.

4.3. Proceso de ejecución

La implementación del programa está dividida en dos partes bien diferenciadas: una encargada de ejecutar el proceso de minado sobre repositorios GitHub y la otra sobre repositorios GitLab. En cada parte se utilizará para su cometido la API correspondiente a los repositorios a los que se les está aplicando el proceso de búsqueda.

En primer lugar se comprueba mediante la variable "execute" si se desea ejecutar o no el proceso en su totalidad. En el caso de que dicha variable sea afirmativa se iniciará el proceso generando las estructuras de datos necesarias que se irán completando con datos obtenidos por el proceso de búsqueda.

Estas estructuras de datos son los "DataFrame" de resultados (matriz de proyectos y sistemas de CI), de lenguajes (matriz de lenguajes y sistemas de CI), de contadores y de estadísticas de datos obtenidos a partir de los ficheros de configuración de estos sistemas (estadísticas de sistemas de CI, lenguajes y escenarios de ejecución de trabajos), las cuales pueden ser generadas de cero o recuperadas a partir de ficheros excel ya generados con anterioridad en función de la variable de configuración "useResultsExcelFile".

A continuación, el programa prosigue ejecutando el proceso de búsqueda sobre

GitHub y acto seguido sobre GitLab, siempre y cuando las variables "doGitHubSearch" y "doGitLabSearch" respectivamente sean positivas.

En cuanto a la parte de ejecución sobre repositorios de la plataforma GitHub, en primer lugar se carga la lista de repositorios que va a ser utilizada para aplicar el heurístico. Esta se puede recuperar mediante la carga desde un fichero binario de Python con extensión ".pickle" o se puede generar desde cero mediante una llamada a la API de GitHub, en función del valor de la variable de configuración "usePickleFile".

Una vez obtenida la lista de repositorios se aplica el heurístico a cada uno de ellos y se irán rellenando las estructuras de datos ya mencionadas con la información que se vaya obteniendo a medida que se ejecuta el proceso. Estas estructuras de datos se devuelven en formato excel en aras de poder ser interpretadas de forma sencilla.

En cuanto a la parte correspondiente a la ejecución sobre proyectos GitLab, al igual que en la parte GitHub ya mencionada, se pueden obtener los repositorios cargados desde un fichero binario de Python con extensión ".pickle" o se pueden recuperar mediante una llamada a la API de GitLab. En el caso de que se tenga que llamar a la API de GitLab para generar los repositorios, se diferencian dos formas de actuar en función de la variable de configuración "search1By1": obteniendo los repositorios y aplicar el heurístico a la lista que se genere, o ir aplicando el heurístico uno a uno según se vaya cargando la lista de repositorios.

Finalmente, se transforman las estructuras de datos de tipo "DataFrame" proporcionadas por la librería "Pandas" en formato Excel o Csv.

A modo de resumen, se expone el flujo de ejecución del programa sobre GitHub y sobre GitLab [C].

4.4. Análisis de ficheros de configuración CI

Para analizar los ficheros de configuración YML de cada herramienta de CI se utiliza el script "ci_yml_parser.py", encargado de transformar la información contenida en dichos ficheros de configuración en objetos Python con los que más tarde se construyen las estructuras de datos de estadísticas de "jobs".

En primer lugar, se va a llamar a la función "makeYMLTmpFile" cuya finalidad es construir en un directorio "tmp" local los diferentes ficheros de configuración YML localizados mediante los literales definidos en la función "getCISearchFiles" del script "ci_tool.py". Si el literal definido como criterio de búsqueda para un sistema de CI en concreto es un directorio, se generarán en local todos los ficheros YML contenidos en dicho directorio, como por ejemplo en el caso de GitHub Actions, en el que se intenta localizar el directorio ".github/workflows".

Una vez generados los ficheros YML en local, se inicia el proceso de conversión de dichos ficheros temporales a objetos Python. Para ello se irán cargando dichos ficheros YML mediante la función "loadData" dando uso a la librería PyYaml que convierte etiquetas YML o YAML en diccionarios Python.

Se utilizan tres funciones diferentes en función de si el sistema de CI tratado es GitHub Actions, Gitlab CI o Travis CI, que van a ser los tres sistemas de CI de los que se van a estudiar aspectos relativos a la configuración de "jobs" automatizados.

En GitLab CI, se contemplan varios aspectos en la disposición de las etiquetas del fichero de configuración YML. En primer lugar se intenta localizar en el primer nivel del fichero YML una etiqueta "stages" en la que se definirán de forma genérica los escenarios en los que se van a ejecutar los "jobs" configurados y, a continuación, se inicia la enumeración de estos "jobs" de acuerdo con los siguientes casos:

- La etiqueta "workflow", se considera como trabajo único cuyos pasos están definidos en la etiqueta "rules" (etiqueta de un nivel inferior a "workflows").

- Las etiquetas definidas en la función "getMainYMLStages".

- Y por último, aquellas etiquetas del nivel superior del fichero YML que contengan una etiqueta "script" en su interior. Estos serán los "jobs" propiamente dichos configurados en el fichero YML, cuyo momento de ejecución se definirá mediante las etiquetas "stage" y "when".

De acuerdo con la documentación de GitLab CI [17] se pueden encontrar dos tipos de escenarios o "stages" según el nombre que se les asigna:

- "stages" **por defecto**:
  - .pre: permite que un trabajo se ejecute al inicio del ciclo.
  - build.
  - test.
  - deploy.
  - .post: permite que un trabajo se ejecute al final del ciclo.
- "stages" **definidos por el usuario**.

El orden de definición de estos "stages" indican el orden en el que se van a ejecutar los "jobs", ejecutándose en paralelo aquellos que pertenezcan al mismo escenario o "stage".

En cuanto a la etiqueta "when" se van a poder encontrar los siguientes casos:

- on_success: ejecuta el "job" cuando todos los de etapas anteriores se han realizado correctamente.
- manual: ejecuta el "job" solamente cuando se especifica manualmente.
- always: ejecuta el "job" independientemente del estado de los trabajos en

etapas anteriores.
- on_failure: ejecuta el "job" cuando falla al menos un trabajo en una etapa anterior.
- delayed: retrasa la ejecución de un "job" durante un tiempo especificado.
- never: no se ejecuta el "job" en ningún caso.

En GitHub Actions es más sencillo el análisis del fichero YML puesto que tiene etiquetas bien definidas para cada aspecto que queremos estudiar. En primer lugar se localiza la etiqueta "on" en la que se definirán los escenarios o "stages" en los que se van a ejecutar los trabajos configurados. A continuación se localiza la etiqueta "jobs", lugar en el que van a ir definidos todos los trabajos automatizados junto con la etiqueta "steps" en la que se enumeran los pasos que van a seguir cada uno de ellos.

Se van a poder encontrar, entre otros, los siguientes "stages" de ejecución de trabajos [18]:

- branch_protection_rule
- check_run
- create
- delete
- deployment
- fork
- gollum
- issue_comment
- pull_request
- push
- release
- schedule
- workflow_dispatch
- ...

En cuanto a Travis CI se contemplan dos casos de configuración de "jobs":

- En el primer caso que nos podemos encontrar, en primer lugar se intenta localizar una etiqueta "stages" genérica con todos los escenarios en los que se van a ejecutar los trabajos. A continuación, al igual que en GitHub Actions, se busca una etiqueta "jobs" y en cada una de sus etiquetas de nivel inferior se intentan localizar las de "stage", "install" y "script", etiquetas en las que se definirán respectivamente los escenarios de ejecución concretos del trabajo así como los pasos que sigue para llevarse a cabo.

- En el segundo caso contemplado, se localizan las etiquetas definidas en en la función "getMainYMLStages".

Por lo tanto, en Travis CI se pueden encontrar "jobs" definidos con la etiqueta "stage" en su interior con valores como "test", "deploy", etc. Pero, en general,

nos vamos a encontrar etiquetas como las que se enumeran a continuación y que están definidas en la función "getMainYMLStages" del script "ci_yml_parser.py" de la herramienta implementada [19]:

- before_install: comandos que se ejecutan antes de los pasos de instalación.
- install: pasos de instalación encargados de preparar el entorno de construcción.
- after_install: comandos que se ejecutan después de los pasos de instalación.
- before_script: comandos que se ejecutan antes de ejecutar las instrucciones que construyen la aplicación.
- script: conjunto de instrucciones encargadas de construir la aplicación.
- after_script: comandos que se ejecutan después de ejecutar las instrucciones que construyen la aplicación.
- before_deploy: instrucciones que se ejecutan antes de la definición del método de despliegue del código.
- deploy: etiqueta en la que se define el método de despliegue del código.
- after_deploy: instrucciones que se ejecutan después de la definición del método de despliegue del código.
- before_cache: instrucciones que se ejecutan antes de guardar en cache información relativa a la aplicación.
- cache: instrucciones que se ejecutan al guardar en cache información relativa a la aplicación.
- after_cache: instrucciones que se ejecutan después de guardar en cache información relativa a la aplicación.
- after_success y after_failure: conjunto de instrucciones que se ejecutan si la ejecución es exitosa o falla.

4.5. Dificultades y problemas encontrados

En la realización de este trabajo se han ido encontrando diferentes errores que han supuesto un impedimento a la hora de continuar.

Los más comunes han sido los siguientes:

- Resulta que la API de GitHub nos permite crear llamadas para obtener información sobre los diferentes repositorios que contiene, y de esta manera poder integrar nuestra aplicación con GitHub, pero con una limitación de 60 peticiones a la hora, número insuficiente para realizar una búsqueda masiva que devuelva una cantidad aceptable de repositorios con la que poder llevar a cabo el estudio de manera objetiva. En el momento en el que se utilizan más de los que esta API puede tratar se lanza la siguiente excepción:

Error 403 - API rate limit exceeded for user ID XXXXXXX

Generando un token de autenticación podemos incrementar el número de peticiones por hora de 60 a 5000 [A]. Sin embargo, este número incrementado de peticiones sigue siendo insuficiente para poder ejecutar el proceso de forma masiva sobre un número muy grande de repositorios.

Como solución, mediante la función "get-rate-limit()" [4.1] de la API de GitHub, antes de gastar una consulta a esta API, se obtiene el número de peticiones restantes y al llegar al límite se pausa el proceso tanto tiempo como sea necesario para recuperar dichas peticiones. Una vez recuperadas las peticiones se prosigue con la ejecución desde donde se dejó en espera.

```
def doApiRateLimitControl():
    try:
        g = authenticate()
        rl = g.get_rate_limit()
        rl_core = rl.core
        core_remaining = rl_core.remaining
        rl_search = rl.search
        search_remaining = rl_search.remaining
        if core_remaining <= 0:
            reset_timestamp = calendar.timegm(rl_core.reset.timetuple())
            sleep_time = reset_timestamp - calendar.timegm(time.gmtime()) + 5
            print("API rate limit exceded: " + str(sleep_time) + " sleep_time. Waiting...")
            time.sleep(sleep_time)
            g = authenticate()
    except:
        aux.printLog("Control del API rate limit exceded NO aplicado...", logging.WARNING)
```

Código 4.1: Control del nº de peticiones a la API GitHub

- La API de GitLab, al igual que la API de GitHub, cuenta con limitaciones a la hora de realizar llamadas sobre ella. En este caso no se limitan el número de peticiones sobre la API, sino que se limitan el número de proyectos que se pueden obtener.

Cada consulta a la API de GitLab cuenta con un número de páginas y en cada página un número X de proyectos, siendo X de 20 a 100 proyectos en función de lo que se configure en la variable "per_page" de la consulta.

Para tratar los repositorios devueltos por una consulta se pueden ir recorriendo las páginas que la conforman y por cada página se pueden ir tratando los repositorios contenidos. Sin embargo, al alcanzar al repositorio número 20.001 del recorrido se lanza la excepción correspondiente a la limitación de GitLab.

Error 405: Offset pagination has a maximum allowed offset of 50000 for requests that return objects of type Project. Remaining records can be retrieved using keyset pagination.

La solución empleada para corregir este problema es la siguiente: en lugar de realizar una única consulta e ir recorriendo las páginas y repositorios de esa única consulta, se generan varias consultas en serie, siempre recorriendo el número máximo de repositorios devueltos en la primera página. Una vez ya han sido los repositorios correspondientes a la primera página de la consulta lanzada, se lanza otra consulta distinta partiendo del identificador del último repositorio tratado en la consulta anterior. De esta forma se puede esquivar la limitación de 20.000 repositorios por consulta impuesta por la API de GitLab e ir obteniendo infinitos proyectos.

- La librería PyYaml permite transformar un fichero con formato YML en un diccionario Python en formato json. Tras ejecutar el proceso de análisis de repositorios de forma masiva se han encontrado casos en los que el fichero YML de configuración del sistema de CI empleado para automatizar trabajos no estaba bien construido, provocando que esta librería no pudiese realizar la conversión de forma satisfactoria, lanzando excepciones similares a la que se muestra a continuación:

while scanning a quoted scalar in "tmp/ftmp_0.yml", line 21, column 27 found unexpected end of stream in "tmp/ftmp_0.yml", line 22, column 1

Por ejemplo, el repositorio "leapsight/bondy" (`https://gitlab.com/leapsight/bondy`), a pesar de haber sido encontrado almacenado en la plataforma GitLab, utiliza la herramienta de integración continua proporcionada por de GitHub Actions en el directorio .github/workflows/docker_image_build.yaml. A simple vista, parece que están bien configuradas las instrucciones de dicho fichero YAML, sin embargo, en la línea 48 del fichero, se observa que hay un guion con un nivel de tabulación distinto al de los demás debido a que tiene un caracter espacio (' ') de más. Esto provoca que la librería PyYaml utilizada no sea capaz de transformar el fichero YML a un objeto diccionario de Python.

No obstante, habría que comprobar si, a pesar de tener ese espacio de más, el fichero de configuración funciona correctamente a la hora de que ejecutar los trabajos que tiene automatizados mediante GitHub Actions en este caso.

- Otro problema relacionados con la API de GitLab es la escasez de filtros que se pueden aplicar sobre cada consulta. A diferencia de la API de GitHub, la de GitLab no permite filtrar por lenguaje o número de estrellas dando lugar a que se tengan que ir comprobando uno a uno mediante los atributos "language" y "stars" de los objetos "project". Esto ha supuesto que el proceso de ejecución sobre proyectos GitLab tarde más en ejecutarse que la ejecución sobre repositorios GitHub.

5

Validación experimental y resultados

5.1. Experimento preliminar

Se realiza un primer experimento ejecutando el programa implementado sobre 500 repositorios de la plataforma GitHub y 500 repositorios de la plataforma GitLab, todos ellos repositorios de código abierto u "open source", sin discriminar entre resultados positivos o negativos en la búsqueda, es decir, entre esos 1000 repositorios se pretende encontrar tanto repositorios que utilizan alguna de las herramientas de integración continua contempladas como repositorios que no las utilizan.

El objetivo de este experimento es conocer cuáles son los sistemas de integración continua más utilizados en repositorios de este tipo para posteriormente analizarlas más en profundidad en un segundo experimento.

Los parámetros con los que se ejecuta el programa son los siguientes:

- Para **GitHub**:
  - created: >2016-01-01
  - pushed: >2021-01-01
  - stars: >=9500
  - forks: >=800
  - archived: false
  - is: public
  - onlyPositives: false
- Para **GitLab**:

- visibility: public
- last_activity_after: 2016-01-01T00:00:00Z
- stars: >25
- onlyPositives: false

Tras lanzar el programa se obtienen los resultados reflejados en la tabla D.1.

A simple vista se puede observar que predomina el uso de los sistemas de integración continua propios de cada plataforma, siendo GitHub Actions para repositorios almacenados en GitHub y GitLab CI para repositorios almacenados en GitLab.

A parte de estos dos sistemas de CI ya mencionados se localizan repositorios que utilizan el sistema de integración continua proporcionado por Travis CI, obteniendo 68 casos en repositorios almacenados en GitHub y otros 25 en repositorios almacenados en GitLab, dando un total de 93 repositorios en ambas plataformas, es decir, en 9'3 % del total de repositorios.

También encontramos repositorios que utilizan Jenkins, que en un principio, se consideró como uno de los sistemas que se iban a utilizar en un mayor número de repositorios junto con GitHub Actions y Travis CI.

Otros sistemas a tener en cuenta son Circle CI, Azure Pipelines y Bazel, obteniendo un total de 42, 6 y 8 repositorios que utilizan dichos sistemas; resultados considerados insuficientes como para profundizar en ellos en el experimento final a realizar.

Finalmente, mencionar también los 2 repositorios encontrados que utilizan la herramienta de integración continua Concourse, uno para GitHub y otro para GitLab, considerados como falsos positivos ya que a pesar de haber cumplido el criterio de búsqueda establecido no se utiliza esta herramienta de integración en ninguno de los dos casos. Esto se debe a que el criterio establecido para la búsqueda de este sistema de CI, "tasks", directorio propuesto por la documentación de Concourse para almacenar los ficheros de configuración de trabajos, es demasiado genérico como para encontrar repositorios que, teniendo ese directorio en su árbol de directorios, usen Concourse.

5.2. Experimento final

Con las conclusiones obtenidas tras la ejecución del primer experimento de este trabajo, se realiza un segundo experimento estudiando más en profundidad los sistemas de CI ofrecidos por Travis CI, GitHub Actions y GitLab CI, ya que son los sistemas de los que se han obtenido más información.

Para ello se modifica el programa de tal forma que se pueda analizar la forma

en la que están construidos los ficheros YML o YAML de configuración de trabajos de las herramientas seleccionadas.

La principal mejora añadida para poder analizar dichos ficheros de configuración es la implementación de un script encargado de transformar cada fichero de configuración CI en objetos Python (ci_yml_parser.py) con el objetivo de poder manejar la información obtenida y construir los ficheros excel de resultados fácilmente.

Este script va a permitir añadir las siguientes columnas al excel principal de resultados:

- **STAGES**: columna en la que se almacenan los escenarios en los que se ejecutan los puntos del fichero de configuración considerados como trabajos.
- **NUM_JOBS**: columna en la que se almacena el número de trabajos que se ejecutan en el repositorio en cuestión.
- **TOTAL_TASKS**: columna en la que se almacena el número total de tareas de todos los trabajos automatizados en el repositorio.
- **TASK_AVERAGE_PER_JOB**: columna en la que se almacena la media de tareas por trabajo.

Con esta información y de manera automática, se generan otros ficheros excel de estadísticas relacionadas con estos ficheros de configuración de trabajos. Estas columnas van contener información en formato json y a modo de diccionario con la intención de diferenciar los datos obtenidos en de cada sistema de CI en particular, ya que puede darse el caso de que se encuentren repositorios que utilicen más de una herramienta de integración continua, que, como se puede ver en los resultados obtenidos, es algo más común de lo que en un principio se podía esperar.

Este segundo experimento consiste en ejecutar el programa tres veces para repositorios GitHub y una vez para repositorios GitLab, encontrando en cada una de estas ejecuciones exclusivamente repositorios con un resultado positivo en la búsqueda, es decir, repositorios que utilizan algún sistema de CI.

Para ello se emplean las siguientes consultas de búsqueda:

- Para **GitHub**:
  - **Ejecución nº 1**: se buscan repositorios GitHub públicos que sean relativamente recientes.
    - created: >2016-01-01
    - pushed: >2021-01-01
    - stars: >=9500
    - forks: >=800
    - archived: false
    - is: public
    - onlyPositives: true

- **Ejecución nº 2**: se buscan repositorios GitHub públicos que hayan sido creados antes de la aparición y auge del sistema de CI GitHub Actions, pero que se hayan seguido modificando recientemente. De esta forma se obtendrán repositorios que podrían haber migrado de algún sistema de CI distinto de las acciones de GitHub a el sistema proporcionado por GitHub, por ejemplo de Jenkins o Travis CI a GitHub Actions.
  - created: 2010-01-01..2016-01-01
  - pushed: >2018-01-01
  - stars: >=10500
  - forks: >=1500
  - archived: false
  - is: public
  - onlyPositives: true
- **Ejecución nº 3**: se buscan repositorios GitHub públicos que hayan sido creados antes de la aparición y auge del sistema de CI GitHub Actions y que se hayan dejado de modificar recientemente.
  - created: 2010-01-01..2016-01-01
  - pushed: <2018-01-01
  - stars: >=500
  - forks: >=250
  - archived: false
  - is: public
  - onlyPositives: true

- Para **GitLab**: se buscan repositorios GitLab de forma genérica.
  - visibility: public
  - last_activity_after: 2016-01-01T00:00:00Z
  - stars: >25
  - onlyPositives: true

Una vez ejecutado el proceso partiendo de las consultas ya mencionadas, se obtienen los resultados comentados a continuación:

- Ejecución nº 1 GitHub y GitLab:

A simple vista se puede observar en la tabla D.2, al igual que en el experimento preliminar, el predominio de sistemas de CI propios de la plataforma en la que se ha ejecutado el programa, es decir, el predominio de GitHub Actions en repositorios almacenados en GitHub y de GitLab CI en repositorios almacenados en GitLab, debido probablemente a la correcta integración y la facilidad de uso de dichos sistemas sobre repositorios almacenados en sus propias plataformas, así como la utilidad de tener tanto el proyecto como la configuración de trabajos automatizados en un mismo sitio.

No obstante, también se encuentran otras herramientas de CI como Travis CI, Circle CI o Azure Pipelines.

Sobre el número de trabajos encontrados y la media de tareas por trabajo calculada en los sistemas de CI GitHub Actions, GitLab CI y Travis CI, datos reflejados en las tablas D.5 y D.10, se sacan conclusiones similares a las ya obtenidas en el experimento preliminar. Sobre repositorios almacenados en GitHub se obtiene un número alto de trabajos en GitHub Actions con una media de 9'57 tareas por trabajo y sobre repositorios almacenados en GitLab, de forma inversamente proporcional a los trabajos encontrados en repositorios GitHub, se obtiene un número elevado de trabajos en GitLab CI, con una media de 9'03 tareas por trabajo. En cuanto a Travis CI se observa que se mantiene estable el número de trabajos, ya sea ejecutando el proceso sobre repositorios GitHub como sobre repositorios GitLab, con una media de entre 2 y 3 tareas por trabajo (2'75 en GitHub y 3'41 en GitLab).

También se puede observar un comportamiento análogo sobre los datos calculados sobre repositorios de GitHub que utilizan GitHub Actions y repositorios de GitLab que utilizan GitLab CI, ya que se obtienen un número similar de repositorios en cada caso, así como el número de trabajos automatizados junto con su media y mediana.

En cuanto a los lenguajes de programación utilizados para implementar los diferentes repositorios a los que se les ha aplicado el proceso, aparecen varios en común situados en la cúspide de los más utilizados, a tener en cuenta, ordenados de mayor a menos uso, los siguientes: Python, Javascript, Go, C++ y Java D.3 - D.4 y D.8 - D.9.

Se observa que Python y JavaScript son los dos lenguajes de programación más utilizados en la implementación de repositorios qur utilizan herramientas de integración continua tanto en GitHub como en GitLab. Estos resultados tendrían explicación en que estos lenguajes de programación son muy utilizados para el desarrollo de aplicaciones web (en el caso de python utilizando el framework Django), que son un tipo de implementación muy propenso a utilizar herramientas de este tipo, sobre todo para la automatización de pruebas. No obstante, el lenguaje de programación Python también se utiliza bastante para el desarrollo del lado del servidor en aplicaciones cliente-servidor.

Además, en cuanto a estos lenguajes de programación, también se obtienen datos estadísticos sobre los trabajos o "jobs" configurados en los ficheros YML de los sistemas de integración continua GitHub Actions, GitLab CI y Travis CI D.6 - D.11.

Otro aspecto a tener en cuenta sobre los datos obtenidos en el proceso de búsqueda son los escenarios en los que se ejecutan los diferentes trabajos automatizados por estos sistemas de CI. En este caso, tal y como se expone en las tablas D.7 y D.12, encontramos diferencias notables, ya que en repositorios almacenados en GitHub predominan escenarios como "push", "pull_request" o "schedule" y en GitLab predominan otros como "build", "test", "deploy" o "release".

- Ejecución nº 2 GitHub:

En esta segunda ejecución sobre repositorios de la plataforma GitHub, a pesar de haber intentado localizar repositorios con herramientas de CI distintas de GitHub Actions buscando solamente aquellos que fueron creados antes de la existencia de dicha herramienta, si se consultan los datos de la tabla D.13 se puede ver como sigue predominando el uso de las acciones de la propia plataforma de GitHub, probablemente debido a que en algún momento de tiempo utilizaban otros sistemas de CI y se migraron recientemente para utilizar GitHub Actions. En trabajos futuros se podrían comprobar "commits" o subidas de código antiguas de estos repositorios para comprobar si esta suposición es o no correcta.

No obstante, en otros sistemas de integración continua como Travis CI o Circle CI se obtienen más datos con respecto a los obtenidos en la primera ejecución de este experimento.

En cuanto a los datos que se obtienen relativos a los lenguajes de programación empleados en cada repositorio, trabajos automatizados, tareas y escenarios en los que son ejecutados dichos trabajos son muy similares a los obtenidos en la primera ejecución de este experimento final. Se sigue observando el predominio de los lenguajes de programación Python y JavaScript, pero esta vez obteniendo más información sobre JavaScript, posicionado en la cúspide de lenguajes de programación más utilizados con un total de 196 repositorios encontrados D.14 - D.15.

- Ejecución nº 3 GitHub:

Se puede observar en los resultados como no aparecen repositorios que usen GitHub Actions, a causa de que los repositorios que se han intentado buscar en esta ejecución son repositorios cuya fecha de creación es anterior al origen de GitHub Actions, y que además, han sido dejados de modificar tras su aparición, es decir, a diferencia de la segunda ejecución de este experimento final sobre proyectos GitHub, no podría darse el caso de que hubiesen migrado de otra herramienta a GitHub Actions.

En la tabla de contadores D.19 queda reflejada la exclusividad de uso de Travis CI para la implementación de la integración continua en proyectos GitHub de código abierto anteriores a la aparición de GitHub Actions, con un total de 130 trabajos y una media de 1'65 tareas por trabajo.

Como se ve reflejado en la tabla D.23, Javascript y Java son los dos lenguajes de programación más utilizados. Esto se debe a que en años anteriores, JavaScript y Java eran los dos lenguajes de programación por excelencia y se utilizaban para prácticamente todo.

En cuanto a los escenarios de ejecución se observa que el programa solamente ha encontrado trabajos sin escenarios definidos en etiquetas dedicadas exclusi-

vamente para ello, como por ejemplo las etiquetas “stage”, “stages”, la etiqueta “when” en el caso de GitLab CI o la etiqueta “on” en el caso de GitHub Actions. Al no encontrar escenarios propiamente dichos, la tabla D.24 se ha rellenado con etiquetas del tipo “script”, “before_install”, “before_script” o “install”.

Además se puede observar como el número total de repositorios y el total de escenarios de cada registro “stage” localizado coinciden, dando a entender como, por lo general, la estructura de los ficheros de configuración de Travis CI en años pasados contenían una etiqueta de cada tipo, pudiendo ser estas etiquetas las siguientes: “before_install”, “install”, “after_install”, “before_script”, “script”, “after_script”, “before_deploy”, “deploy”, “after_deploy”, “before_cache”, “cache”, “after_cache”, “after_success” y/o “after_failure”.

5.3. Resumen de resultados

En esta sección, a modo de resumen del análisis llevado a cabo en los dos experimentos ejecutados, se va a proceder a contestar aquellas preguntas que se expusieron al inicio de este estudio:

- ¿Cuáles son los sistemas de CI más utilizados en cada plataforma?

Se observa como en cada plataforma predominan sus propios servicios de integración continua. En el caso de GitHub el sistema GitHub Actions y en el caso de GitLab el sistema GitLab CI.

- ¿Se suelen combinar más de un sistema de CI en un mismo repositorio?

Sí, se suelen combinar más de un sistema de CI, para que cada uno de ellos lleve a cabo tareas diferentes o por que se esté migrando de un sistema de CI a otro y estén conviviendo juntos en algún determinado momento.

En el experimento preliminar, en el caso de GitHub encontramos 55 repositorios que utilizan más de un sistemas de CI, siendo un 11 % de los 500 repositorios totales y un 17’74 % de los 310 repositorios con al menos un sistema de CI en uso. En el caso de GitLab, encontramos 31 repositorios con más de un sistema de CI, siendo el 6’2 % de los 500 repositorios totales y el 10’4 % de los que utilizan sistemas de CI exclusivamente.

En cuanto al experimento final, en la primera ejecución encontramos 92 y 45 repositorios, en GitHub y GitLab respectivamente, que utilizan más de un sistema de CI, siendo el 16’2 % de los 568 totales en GitLab y el 8’96 % de los 502 totales en GitLab. En la segunda ejecución, ejecutada exclusivamente para repositorios almacenados en GitHub, encontramos 146 repositorios que utilizan más de un sistema de CI, siendo el 21’66 % de los 674 repositorios totales. En la

tercera y última ejecución, ejecutada también solamente para repositorios alojados en GitHub, se obtienen exclusivamente 79 repositorios que utilizan el servicio proporcionado por Travis CI, dando lugar a que sea el 0 % los repositorios que utilizan más de un sistema de CI.

- ¿Qué lenguajes de programación predominan en repositorios que utilizan sistemas de CI?

Los lenguajes que más aparecen en cuanto al uso de sistemas de integración continua en proyectos software son Python y JavaScript, independientemente de la plataforma en la que estén alojados los repositorios (GitHub o GitLab).

- ¿Cuántos "jobs" se suelen configurar en repositorios de código abierto?

El número de trabajos que suele haber en proyectos "open source", viendo los resultados obtenidos se observa que depende del sistema de CI que los configura. En el experimento final, primera ejecución, se obtiene ejecutando el proceso para repositorios almacenados en GitHub un número medio de 2'75, 9'57 y 12'25 trabajos en Travis CI, GitHub Actions y GitLab CI respectivamente. Estos datos se corresponden con una media total de 8'19 trabajos en proyectos de código abierto. Para esta misma ejecución pero sobre repositorios almacenados en GitLab, se obtiene una media de 3'41, 4'70 y 9'03 trabajos en Travis CI, GitHub Actions y GitLab CI, siendo la media total de 5'71 trabajos por repositorio.

En la segunda ejecución, que solamente se ejecuta el proceso sobre repositorios almacenados en GitHub, se obtiene una media de trabajos por repositorio de 3'41 en Travis CI, 7'46 en GitHub Actions y 4'89 en GitLab CI, dando lugar a una media total de 5'25 trabajos por repositorios.

Por último, la tercera ejecución, en la que solamente se obtienen repositorios que utilizan Travis CI con una media de trabajos por repositorio de 1'65.

- ¿Cuáles son los principales escenarios o "stages" en los que se suelen ejecutar trabajos automatizados?

En cuanto a los principales escenarios de ejecución de trabajos en proyectos "open source" se podrían destacar "push" y "pull_request" en el caso de GitHub Actions, "build" y "test" en el caso de GitLab CI y la etiqueta "script" en el caso de Travis CI.

6

Conclusiones y trabajos futuros

La realización de este estudio me ha permitido, en primer lugar, ampliar mis conocimientos sobre las diferentes herramientas de integración continua existentes y conocer cuáles son las predominantes en entornos de alojamiento de proyectos software como GitHub y GitLab. También he ampliado conocimientos en técnicas de "research" mediante la minería de repositorios utilizando las APIs de GitHub y GitLab para implementar un programa en Python, que es uno de los lenguajes que más está creciendo actualmente. Con 11.3 millones de usuarios, en el año 2021, Python ha sido, en cuanto a crecimiento, uno de los lenguajes de programación más a tener en cuenta, consiguiendo precisamente ocupar este vacío como favorito en desarrollo DS/ML, pero además siendo una opción para desarrollo de aplicaciones de IoT (internet de las cosas).

En primer lugar se estudiaron diversas herramientas de integración continua, tanto su funcionamiento como la estructura de los ficheros de configuración de trabajos automatizados con el objetivo de implementar un heurístico que fuese capaz de encontrar repositorios que usasen herramientas de este tipo.

A continuación se realizaron varios experimentos aplicando el heurístico, de los que se obtuvieron resultados suficientes como para poder tomar conclusiones al respecto.

Tras la realización de estos experimentos se puede destacar cómo cada vez es más habitual que los programadores que integran sus proyectos de forma continua utilicen herramientas propias de la plataforma en donde están alojando sus proyectos, en este caso GitHub Actions si el repositorio está en GitHub o GitLab CI en el caso de que se almacene en GitLab, probablemente debido a la fácil in-

tegración que puedan tener con sus proyectos al estar almacenados en su propia plataforma.

No obstante, también se siguen usando otros medios para poder emplear integración continua como puedan ser Travis o incluso Jenkins que en un principio parecía que iban a ser más habituales en proyectos de código abierto GitHub o GitLab.

Además, se puede apreciar como en muchos de los repositorios obtenidos en el proceso utilizan más de una herramienta de integración continua, por lo general, una clásica como pueden ser las ya mencionadas Travis y Jenkins, y otra moderna, siendo esta última la propia de la plataforma. Esto se podría deber a que aún hay proyectos tratando de transitar de una herramienta de CI a otras más actuales.

En cuanto a los trabajos futuros, se intentarán ejecutar de algún modo los trabajos definidos en cada fichero YML o YAML de configuración y replicar tests automatizados para comprobar su efectividad. Además, se revisarán en cada repositorio subidas de código o "commits" antiguos para comprobar si en los inicios de los repositorios se utilizaban sistemas de integración continua distintos al que tengan actualmente, es decir, comprobar si se produjo algún tipo de migración de sistemas y con qué objetivo. Por último, mencionar que se pondrán los resultados a disposición de investigadores interesados en estudiar otros aspectos sobre integración continua.

Bibliografía

[1] RedHat, "¿qué son la integración y la distribución continuas (ci/cd)?" 2018. [Online]. Available: https://www.redhat.com/es/topics/devops/what-is-ci-cd

[2] A. G. Paul M. Duvall, Steve Matyas, *Continuous Integration Improving Software Quality and Reducing Risk*. The Addison Wesley Signature Series, 2007.

[3] RedHat, "El concepto de devops," 2018. [Online]. Available: https://www.redhat.com/es/topics/devops

[4] NetApp, "¿qué es devops?" [Online]. Available: https://www.netapp.com/es/devops-solutions/what-is-devops/

[5] *Jenkins*. [Online]. Available: https://www.jenkins.io/doc/book/pipeline/jenkinsfile/

[6] *Travis CI*. [Online]. Available: https://docs.travis-ci.com/user/tutorial/

[7] *Circle CI*. [Online]. Available: https://circleci.com/docs/2.0/about-circleci/?section=getting-started

[8] *GitHub Actions*. [Online]. Available: https://docs.github.com/es/actions

[9] *Azure Pipelines*. [Online]. Available: https://docs.microsoft.com/es-es/azure/devops/pipelines/get-started/pipelines-get-started?view=azure-devops

[10] *Atlassian Bamboo*. [Online]. Available: https://confluence.atlassian.com/bamboo/bamboo-yaml-specs-938844479.html

[11] *Concourse*. [Online]. Available: https://concourse-ci.org/docs.html

[12] *Gitlab CI*. [Online]. Available: https://docs.gitlab.com/ee/ci/pipelines/index.html

[13] *Codeship*. [Online]. Available: https://www.cloudbees.com/products/codeship

[14] *TeamCity*. [Online]. Available: https://www.jetbrains.com/help/teamcity/create-pipeline.html

[15] *Bazel*. [Online]. Available: https://docs.bazel.build/versions/4.2.1/guide.html

[16] *Semaphore CI*. [Online]. Available: https://docs.semaphoreci.com/guided-tour/getting-started/

[17] *Events that trigger workflows*. [Online]. Available: https://docs.github.com/en/actions/using-workflows/events-that-trigger-workflows

[18] *Keyword reference for GitLab "stages"*. [Online]. Available: https://docs.gitlab.com/ee/ci/yaml/#stages

[19] E. Krout, "Anatomy of a travis ci file," *A Cloud Guru*, 2021. [Online]. Available: https://acloudguru.com/blog/engineering/anatomy-of-a-travis-ci-file

Apéndices

A

Minería de repositorios GitHub

A.1. Librería Python "PyGithub"

Para poder llevar a cabo sobre la plataforma GitHub la técnica de research conocida como minería de repositorios, MSR por sus siglas en inglés (Mining Software Repositories), se utiliza la biblioteca "PyGithub", la cual nos va a permitir manejar diferentes recursos de GitHub como repositorios, perfiles de usuario, organizaciones, etc. desde cualquier script Python.

Para instalar la biblioteca bastaría con ejecutar el comando pip install pygithub o clonarlo directamente desde el propio GitHub.

Una vez instalado ya se podría importar desde cualquier script Python mediante la siguiente instrucción:

```
1    from github import Github
```

Con la biblioteca ya importada en el script podemos generar un objeto "GitHub" y, partiendo de ese objeto, realizar consultas sencillas utilizando una serie de parámetros definidos por la librería.

```
1    usuario = "<usuario>"
2    token = "<token>"
3    g = Github(usuario, token)
```

Código A.1: Autenticación en API de GitHub

Al ejecutar una consulta, no se realiza ninguna consulta o búsqueda como tal, sino que se obtiene un objeto generador y se comienza a ejecutar la consulta al iterar sobre dicho objeto generador.

```
1    query= "<query>"
2    generator=g.search_repositories(query=query)
```

Listando el objeto generador "generator" devuelto por la función "search_repositories" de la API de GitHub, se obtienen todos los repositorios que satisfacen la query pasada como parámetro. Esta lista de repositorios va a ser una lista de objetos "Repository" de los que se van a poder obtener una infinidad de información relativa a cada uno de ellos.

A.2. Generación de token de autenticación GitHub

Generar un token de autenticación GitHub nos permite aumentar el número de peticiones por hora que disponemos, pasando de tener 60 a 1500. Este token de autenticación lo podemos generar accediendo a los ajustes de desarrollador de GitHub siguiendo los siguientes pasos:

1. Desplegar ajustes de usuario pinchando en el icono redondo situado en la parte superior derecha de la pantalla y acceder a la opción "Settings".

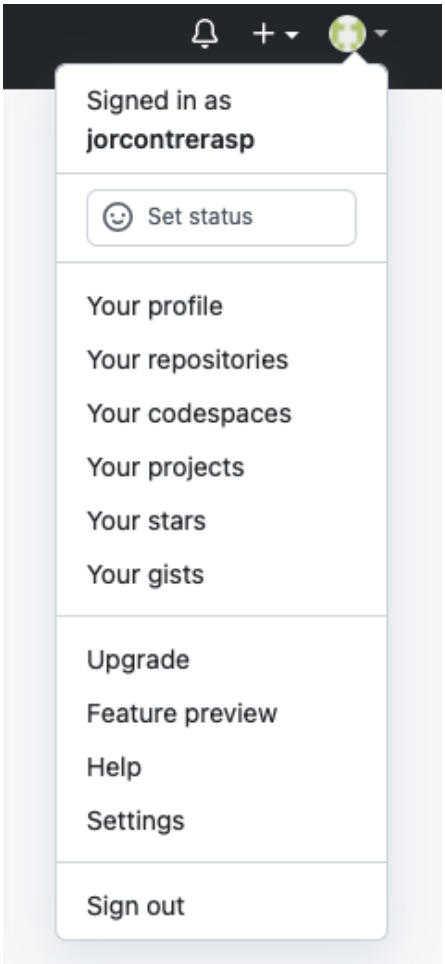

Figura A.1: Generar token GitHub. Paso 1.

2. Una vez dentro de "Settings", acceder a la opción de menú "Developer settings".

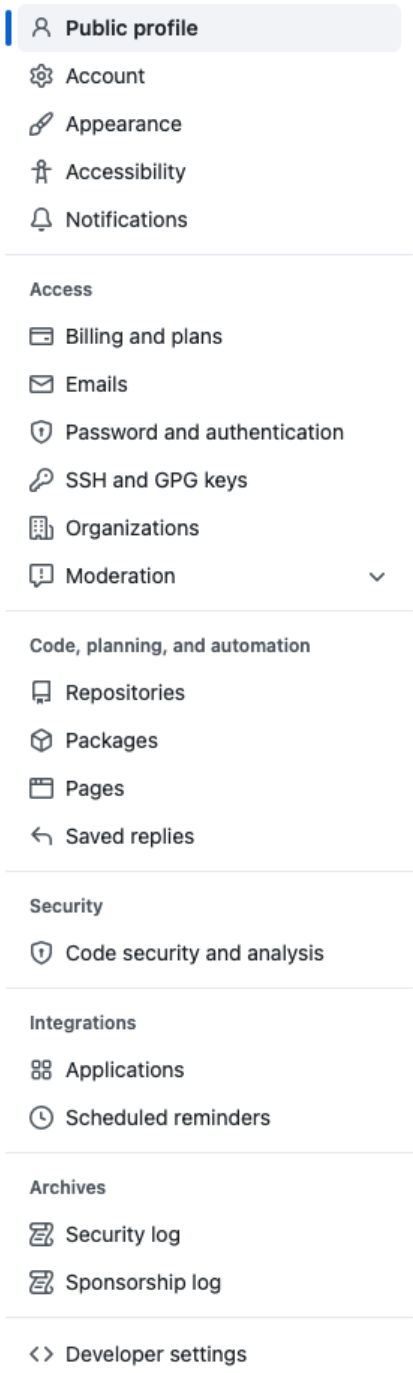

Figura A.2: Generar token GitHub. Paso 2.

3. Accediendo a la opción “Personal Access tokens” y pinchando el botón “Generate new token” nos permitirá rellenar los datos relacionados con el token que utilizaremos posteriormente para autenticarnos utilizando la API de GitHub.

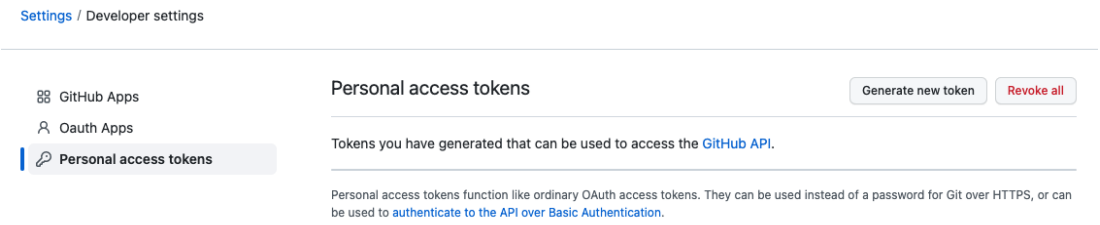

Figura A.3: Generar token GitHub. Paso 3.

4. Rellenar un nombre y seleccionar los permisos del token a generar. Para este caso seleccionar la opción “repo” sería más que suficiente.

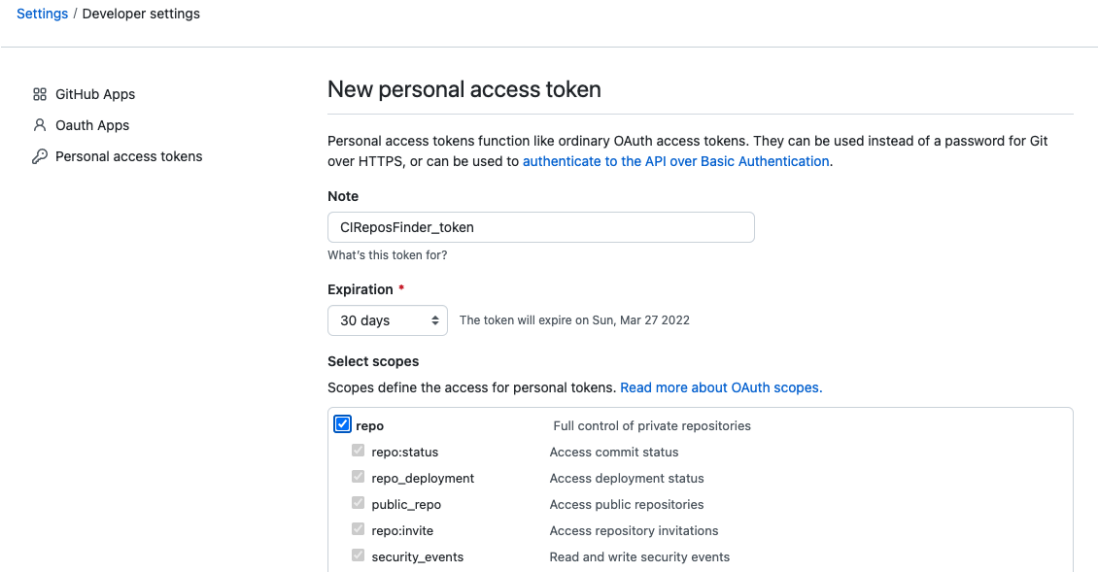

Figura A.4: Generar token GitHub. Paso 4.

5. Por último, pinchar en el botón de generar token.

Figura A.5: Generar token GitHub. Paso 5.

B

Minería de repositorios GitLab

B.1. Librería Python "Gitlab"

Minería de repositorios también se puede llevar a cabo sobre la plataforma de almacenamiento de software GitLab. Para ello es necesario utilizar la librería Python que ofrece la propia plataforma GitLab, conocida con el nombre de "Python-GitLab".

Una vez instalada la librería mediante el comando de instalación de paquetes desde un terminal (pip install python-gitlab), se puede importar en el proyecto de la siguiente manera:

```
1     import gitlab
```

Una vez importada la biblioteca, se puede generar un objeto de la clase "GitLab" que se utilizará para listar proyectos en base a un conjunto de criterios pasados como parámetros de búsqueda. Este objeto se construye de la siguiente forma:

```
1     token = aux.readFile("tokens/gitlab_token.txt")
2     gl = gitlab.Gitlab('http://gitlab.com', private_token=token)
```

Código B.1: Autenticación en API de GitLab

A partir de este objeto ya generado, se pueden listar proyectos:

```
1     projects = gl.projects.list(visibility='public',
2                                 last_activity_after='2016-01-01
    T00:00:00Z',
3                                 pagination='keyset',
4                                 id_after=idAfter,
5                                 page=1,
6                                 order_by='id',
7                                 sort='asc')
```

En este caso se buscarán proyectos de código abierto o públicos cuya última actividad es posterior al año 2016, partiendo desde el identificador de proyecto "idAfter" pasado como parámetro y ordenados por identificador de forma ascendente.

B.2. Generación de token de autenticación GitLab

Para poder acceder a la API que proporciona la plataforma GitLab y poder obtener información sobre los diferentes proyectos de código abierto "open source" es necesario generar un token de autenticación de la siguiente manera:

1. Desplegar ajustes de usuario pinchando en el icono situado en la parte superior derecha de la pantalla y acceder a la opción "Preferences".

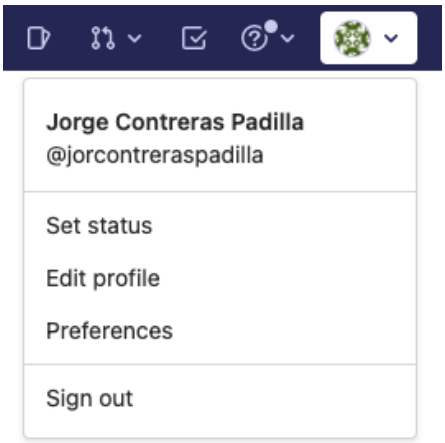

Figura B.1: Generar token GitHub. Paso 1.

2. En el desplegable situado a la izquierda de la pantalla, acceder a la opción de menú "Access Tokens".

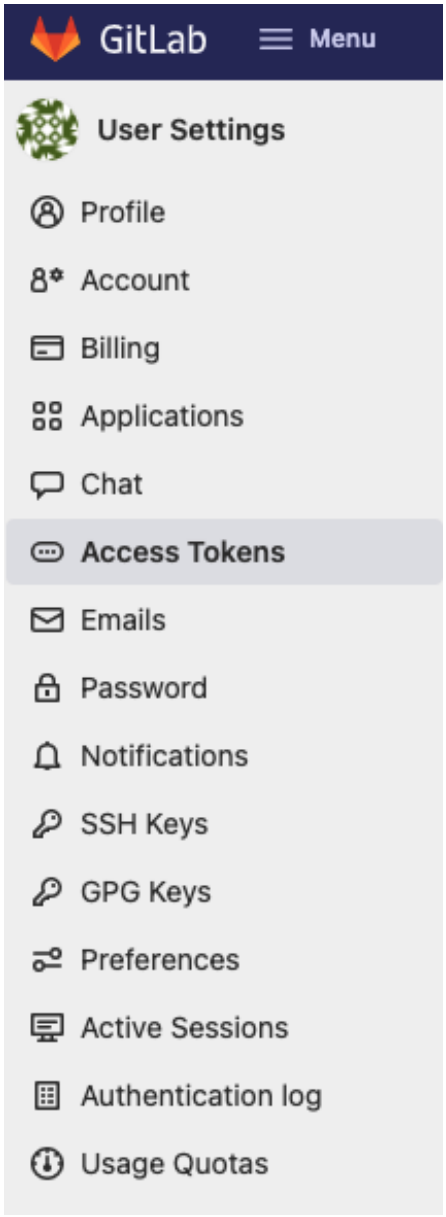

Figura B.2: Generar token GitHub. Paso 2.

3. A continuación aparecerá una pantalla para rellenar los campos necesarios para la generación del token de autenticación: nombre del token, fecha de validez y permisos.

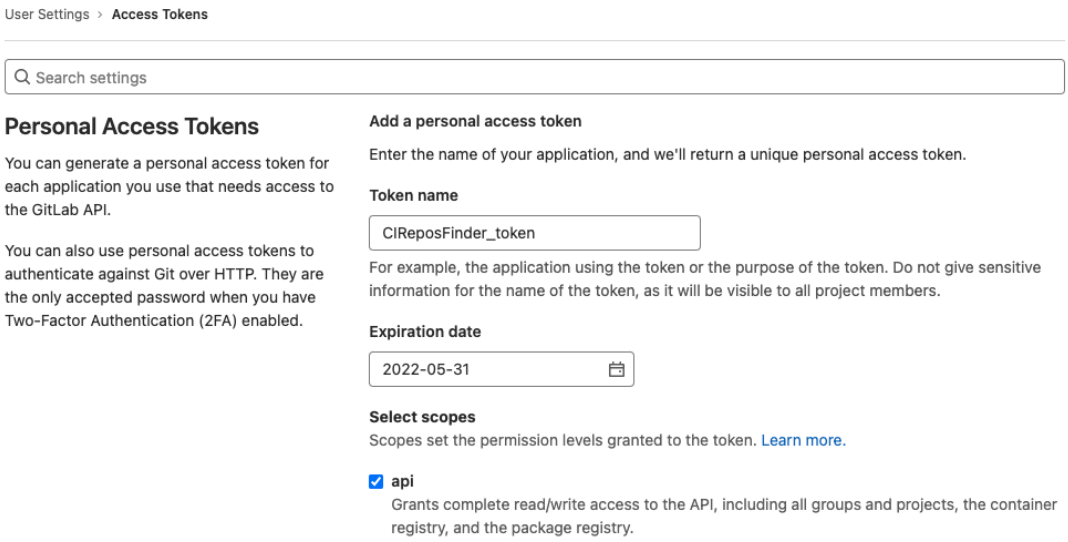

Figura B.3: Generar token GitHub. Paso 3.

4. Por último, una vez rellenos los campos de la pantalla con la información requerida, pinchar en el botón “Create personal access token” para generar el token de autenticación.

Figura B.4: Generar token GitHub. Paso 4.

C

Diagramas de flujo de ejecución de la herramienta

En esta sección se exponen los diagramas de flujo de ejecución de la herramienta implementada, tanto para la parte en la que se ejecuta el proceso sobre repositorios de la plataforma GitHub como para la parte en la que se ejecuta el proceso sobre repositorios de la plataforma GitLab.

C.1. Flujo de ejecución GitHub

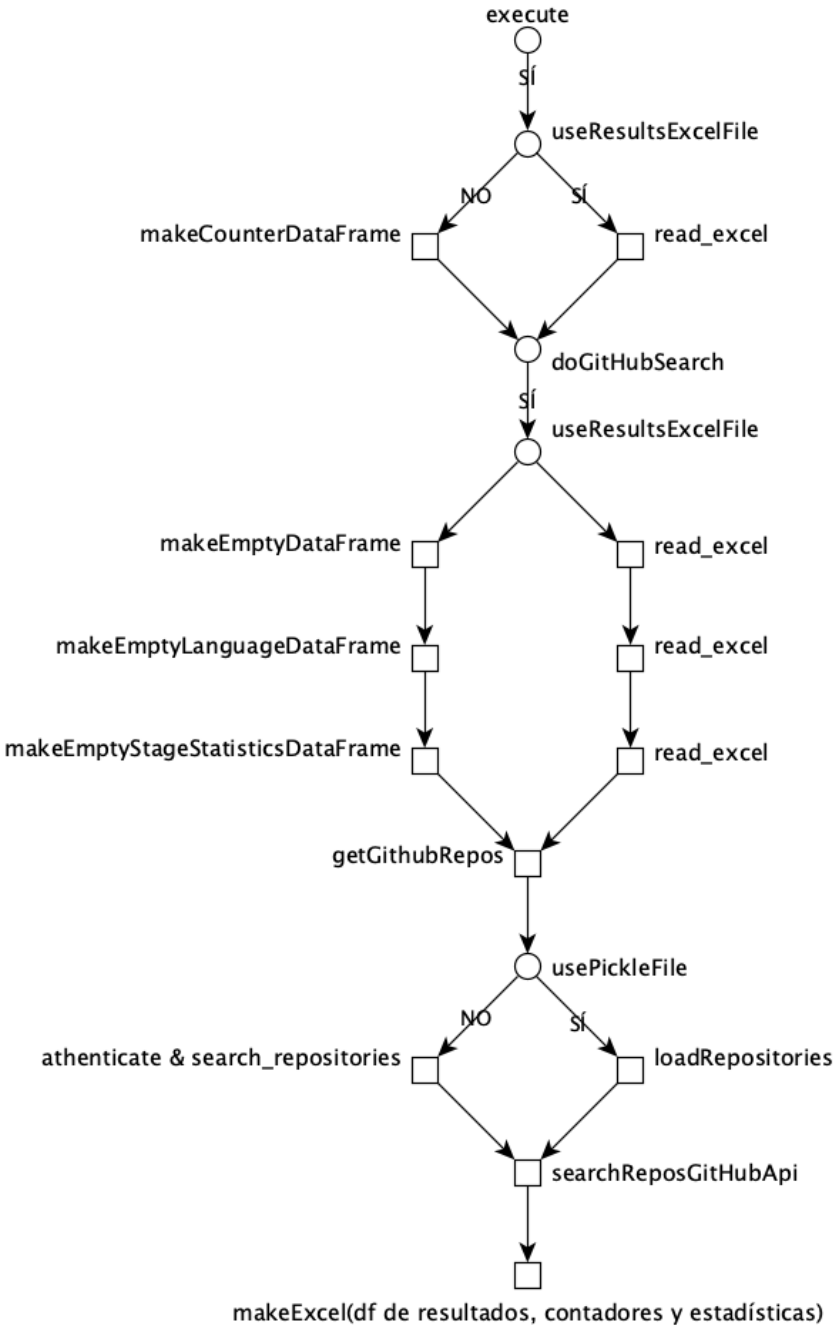

Figura C.1: Flujo de ejecución GitHub.

C.2. Flujo de ejecución GitLab

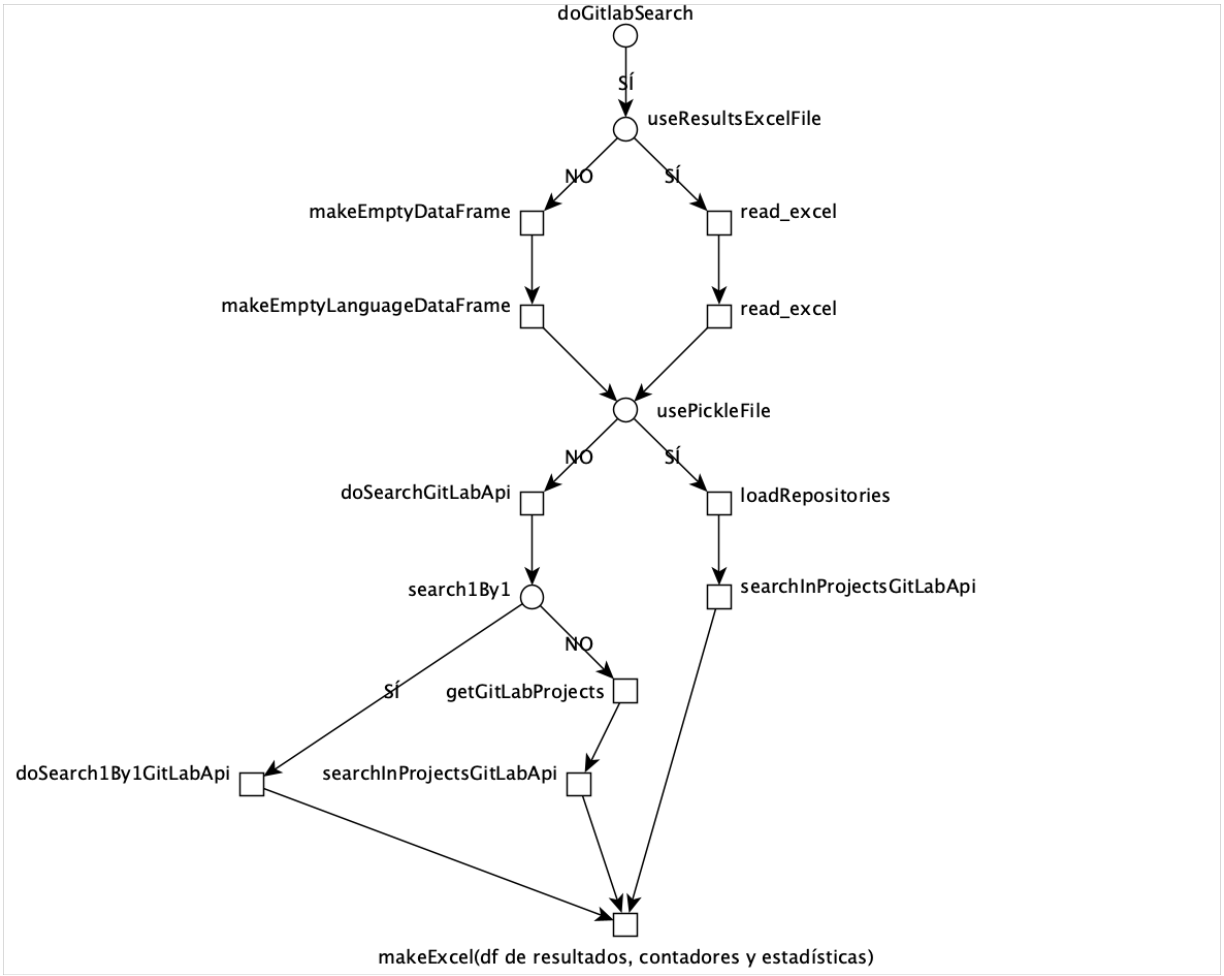

Figura C.2: Flujo de ejecución GitLab.

D

Tablas de resultados

En esta sección se exponen las tablas de resultados obtenidas tras ejecutar el proceso en los diferentes experimentos realizados. Se pueden consultar más en detalle en el directorio "final_results" del repositorio `https://github.com/jorcontrerasp/CIReposFinder`.

D.1. Experimento preliminar

Tabla D.1: Exp. preliminar. Contadores.

	Encontrados GitHub	Encontrados GitLab
jenkins	5	0
travis	68	25
circle ci	40	2
github actions	242	11
azure pipelines	5	1
bamboo	0	0
concourse	1	1
gitlab ci	0	287
codeship	0	0
teamcity	1	0
bazel	7	1
semaphore ci	0	0
appveyor	0	0
Totales	310	296

D.2. Experimento final

D.2.1. Ejecución nº 1 (GitHub y GitLab)

Tabla D.2: Exp. final, ejecución 1. Contadores.

	Encontrados GitHub	Encontrados GitLab
jenkins	7	1
travis	118	37
circle ci	58	4
github actions	458	20
azure pipelines	9	1
bamboo	1	0
gitlab ci	4	487
codeship	0	0
teamcity	1	0
bazel	10	1
semaphore ci	0	0
appveyor	0	0
Totales	568	502

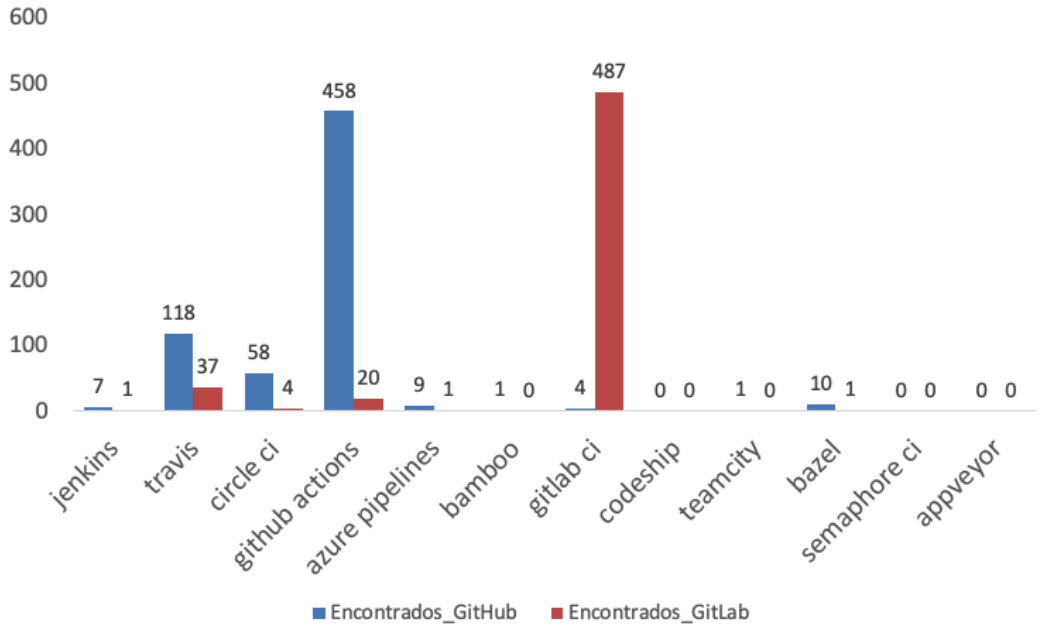

Figura D.1: Exp. final, ejecución 1. Contadores.

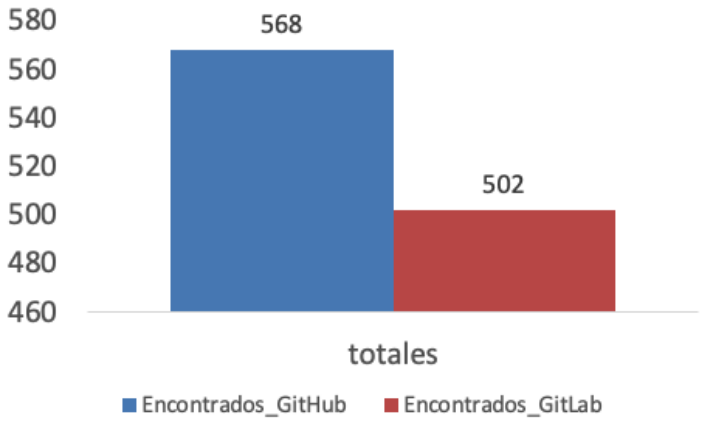

Figura D.2: Exp. final, ejecución 1. Contadores.

Tabla D.3: Exp. final, ejecución 1. Lenguajes en GitHub.

	Jenkins	Travis	Circle CI	GitHub Actions	Azure Pipelines	Bamboo
javascript	1	28	9	75	3	0
python	4	18	11	68	0	0
typescript	0	6	15	77	1	0
go	0	13	4	56	0	1
java	0	18	4	30	0	0
c++	1	5	3	26	4	0
...	...	...	...	...	...	...
Totales	7	118	58	458	9	1

Tabla D.4: Exp. final, ejecución 1. Lenguajes en GitHub (continuación).

	GitLab CI	Codeship	TeamCity	Bazel	Semaphore CI	AppVeyor	TOTALES
javascript	0	0	0	0	0	0	116.0
python	1	0	0	2	0	0	104.0
typescript	0	0	1	2	0	0	102.0
go	0	0	0	1	0	0	75.0
java	1	0	0	0	0	0	53.0
c++	2	0	0	5	0	0	46.0
...	...	...	...	...	...	...	...
Totales	4	0	1	10	0	0	-

Tabla D.5: Exp. final, ejecución 1. Estadísticas de CI GitHub.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
travis	118	324	1	15	2.75	2
github actions	458	4382	1	546	9.57	4
gitlab ci	4	49	1	45	12.25	2

Tabla D.6: Exp. final, ejecución 1. Estadísticas por lenguaje GitHub.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
javascript	103	525	1	73	5.10	3.0
python	87	669	1	78	7.69	4.0
typescript	83	699	1	59	8.42	5.0
go	63	497	1	57	7.89	5.0
java	42	227	1	27	5.40	3.0
c++	36	1151	1	546	31.97	2.0

Tabla D.7: Exp. final, ejecución 1. Estadísticas de escenarios CI GitHub.

	Total proyectos	Total escenarios
push[github actions]	421	3037
pull_request[github actions]	370	2119
schedule[github actions]	196	584
workflow_dispatch[github actions]	151	1272
script[travis]	99	105
release[github actions]	61	164

Tabla D.8: Exp. final, ejecución 1. Lenguajes en GitLab.

	Jenkins	Travis	Circle CI	GitHub Actions	Azure Pipelines	Bamboo
python	0	5	0	1	0	0
c	0	7	0	8	0	0
javascript	0	5	0	0	0	0
c++	0	3	1	6	1	0
ruby	0	2	0	0	0	0
rust	0	3	0	1	0	0
go	0	0	0	0	0	0
java	0	1	0	0	0	0
...	...	...	...	...	...	...
Totales	1	37	4	20	1	0

Tabla D.9: Exp. final, ejecución 1. Lenguajes en GitLab (continuación).

	GitLab CI	Codeship	TeamCity	Bazel	Semaphore CI	AppVeyor	TOTALES
python	82	0	0	0	0	0	88.0
c	48	0	0	0	0	0	63.0
javascript	51	0	0	0	0	0	56.0
c++	42	0	0	0	0	0	53.0
ruby	33	0	0	0	0	0	35.0
rust	27	0	0	0	0	0	31.0
go	29	0	0	0	0	0	29.0
java	26	0	0	0	0	0	27.0
...	...	...	...	...	...	...	...
Totales	487	0	0	1	0	0	-

Tabla D.10: Exp. final, ejecución 1. Estadísticas de CI GitLab.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
travis	37	126	1	11	3.41	3
github actions	20	94	1	42	4.70	1
gitlab ci	487	4398	1	716	9.03	4

Tabla D.11: Exp. final, ejecución 1. Estadísticas por lenguaje GitLab.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
python	84	1294	1	716	15.40	5.0
javascript	53	236	1	18	4.45	3.0
c	51	509	1	62	9.98	7.0
c++	45	689	1	121	15.31	7.0
ruby	33	353	1	97	10.70	4.0
go	29	212	1	33	7.31	4.0
rust	27	219	1	40	8.11	5.0
java	26	142	1	19	5.46	4.5

Tabla D.12: Exp. final, ejecución 1. Estadísticas de escenarios CI GitLab.

	Total proyectos	Total escenarios
build[gitlab ci]	200	696
test[gitlab ci]	197	1003
script[gitlab ci]	145	500
before_script[gitlab ci]	134	134
deploy[gitlab ci]	129	318
cache[gitlab ci]	80	80
script[travis]	34	43
release[gitlab ci]	32	52

D.2.2. Ejecución nº 2 (GitHub)

Tabla D.13: Exp. final, ejecución 2. Contadores.

	Encontrados GitHub
jenkins	14
travis	206
circle ci	75
github actions	512
azure pipelines	14
bamboo	0
gitlab ci	9
codeship	0
teamcity	2
bazel	13
semaphore ci	0
appveyor	0
Totales	674

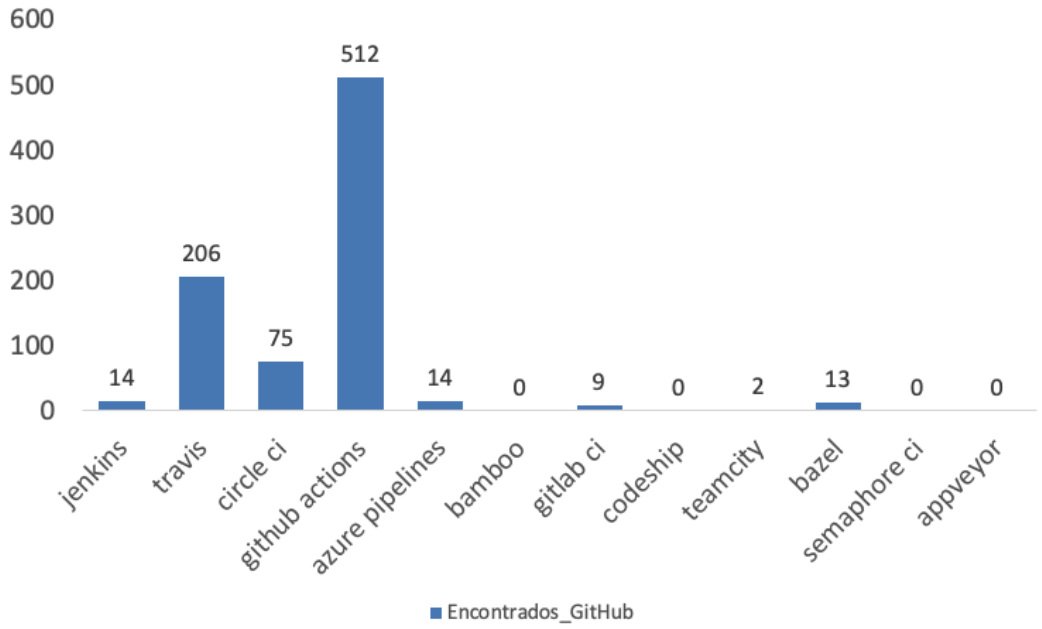

Figura D.3: Exp. final, ejecución 2. Contadores.

Tabla D.14: Exp. final, ejecución 2. Lenguajes.

	Jenkins	Travis	Circle CI	GitHub Actions	Azure Pipelines	Bamboo
javascript	0	60	17	115	1	0
python	0	18	9	57	5	0
java	5	26	4	52	1	0
go	3	7	14	53	1	0
typescript	2	5	10	47	1	0
c++	1	13	4	42	1	0
c	0	12	3	19	1	0
...	...	...	...	...	...	...
Totales	14	206	75	512	14	0

Tabla D.15: Exp. final, ejecución 2. Lenguajes (continuación).

	GitLab CI	Codeship	TeamCity	Bazel	Semaphore CI	AppVeyor	TOTALES
javascript	2	0	1	0	0	0	196.0
python	1	0	0	1	0	0	91.0
java	1	0	0	2	0	0	91.0
go	1	0	1	2	0	0	82.0
typescript	0	0	0	3	0	0	68.0
c++	0	0	0	5	0	0	66.0
c	0	0	0	0	0	0	35.0
...	...	...	...	...	...	...	...
Totales	9	0	2	13	0	0	-

Tabla D.16: Exp. final, ejecución 2. Estadísticas de CI GitHub.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
travis	206	702	1	97	3.41	2
github actions	512	3822	1	312	7.46	4
gitlab ci	9	44	1	9	4.89	4

Tabla D.17: Exp. final, ejecución 2. Estadísticas por lenguaje GitHub.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
javascript	169	822	1	43	4.86	3.0
python	70	516	1	44	7.37	5.0
java	67	449	1	97	6.70	4.0
go	60	588	1	109	9.80	6.0
typescript	53	445	1	35	8.40	4.0
c++	48	336	1	25	7.00	5.5

Tabla D.18: Exp. final, ejecución 2. Estadísticas de escenarios CI GitHub.

	Total proyectos	Total escenarios
push[github actions]	456	2337
pull_request[github actions]	448	2196
schedule[github actions]	207	620
script[travis]	153	184
workflow_dispatch[github actions]	148	948
install[travis]	77	99

D.2.3. Ejecución nº 3 (GitHub)

Tabla D.19: Exp. final, ejecución 3. Contadores.

	Encontrados GitHub
jenkins	0
travis	79
circle ci	0
github actions	0
azure pipelines	0
bamboo	0
gitlab ci	0
codeship	0
teamcity	0
bazel	0
semaphore ci	0
appveyor	0
Totales	79

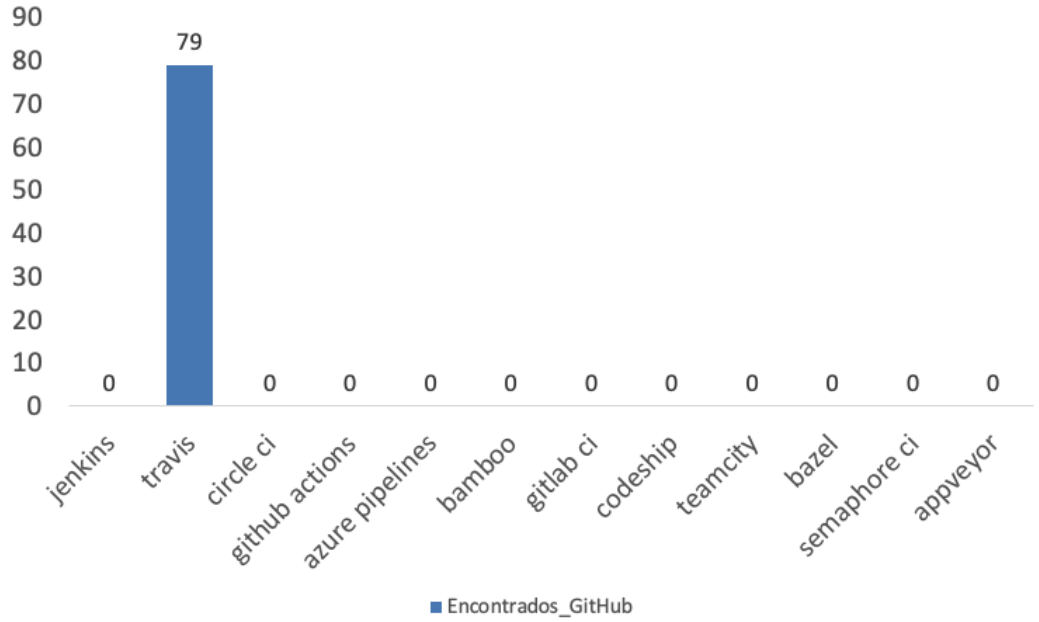

Figura D.4: Exp. final, ejecución 3. Contadores.

Tabla D.20: Exp. final, ejecución 3. Lenguajes.

	Jenkins	Travis	Circle CI	GitHub Actions	Azure Pipelines	Bamboo
javascript	0	34	0	0	0	0
java	0	11	0	0	0	0
objective-c	0	9	0	0	0	0
ruby	0	6	0	0	0	0
...	...	...	...	...	...	...
TOTALES	0	79	0	0	0	0

Tabla D.21: Exp. final, ejecución 3. Lenguajes (continuación).

	GitLab CI	Codeship	TeamCity	Bazel	Semaphore CI	AppVeyor	TOTALES
javascript	0	0	0	0	0	0	34.0
java	0	0	0	0	0	0	11.0
objective-c	0	0	0	0	0	0	9.0
ruby	0	0	0	0	0	0	6.0
...	...	...	...	...	...	...	...
TOTALES	0	0	0	0	0	0	-

Tabla D.22: Exp. final, ejecución 3. Estadísticas de CI GitHub.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
travis	79	130	1	9	1.65	1
github actions	0	0	0	0	0.00	0
gitlab ci	0	0	0	0	0.00	0

Tabla D.23: Exp. final, ejecución 3. Estadísticas por lenguaje GitHub.

	Total repositorios	Total trabajos	Min	Max	Media	Mediana
javascript	34	40	1	4	1.18	1.0
java	11	21	1	3	1.91	2.0
objective-c	9	15	1	3	1.67	2.0
ruby	6	10	2	3	1.67	2.0

Tabla D.24: Exp. final, ejecución 3. Estadísticas de escenarios CI GitHub.

	Total repositorios	Total escenarios
script[travis]	43	43
before_install[travis]	28	28
before_script[travis]	21	21
install[travis]	14	14
cache[travis]	12	12
after_success[travis]	4	4