
Universidad Politécnica
de Madrid

**Escuela Técnica Superior de Ingenieros Informáticos**


Grado en Ingeniería Informática

Trabajo Fin de Grado

# Arquitectura de Testing para la Integración de Sistemas Multi-Agente


Autor: Pablo González Abad

Tutores: Jose María Barambones Ramírez y Ricardo Imbert Paredes



Madrid, Junio 2024

Este Trabajo Fin de Grado se ha depositado en la ETSI Informáticos de la Universidad Politécnica de Madrid para su defensa.

*Trabajo Fin de Grado*
*Grado en Ingeniería Informática*
*Título:* Arquitectura de Testing para la Integración de Sistemas Multi-Agente
Junio 2024

*Autor:* Pablo González Abad

*Tutores:*

Jose María Barambones Ramírez
Departamento de LENGUAJES Y SISTEMAS INFORMÁTICOS E INGENIERÍA DE SOFTWARE (DLSIIS)
ETS de Ingenieros Informáticos
Universidad Politécnica de Madrid

Ricardo Imbert Paredes
Departamento de LENGUAJES Y SISTEMAS INFORMÁTICOS E INGENIERÍA DE SOFTWARE (DLSIIS)
ETS de Ingenieros Informáticos
Universidad Politécnica de Madrid

# Resumen

Este proyecto se basa en el estudio, desarrollo e implementación de una infraestructura inicial y estructura de pruebas para la práctica World of Agent.

World of Agents es la segunda práctica del curso de Desarrollo de Software Basado en Agentes del prestigioso Máster Europeo en Ingeniería de Software en la cual desean crear los jugadores artificiales, los agentes, del videojuego propuesto.

Para la elaboración del trabajo de fin de grado, se han utilizado para la infraestructura la tecnología Gitlab CI/CD junto a la creación de runners y la tecnología Docker. Por otro lado, las tecnologías de Jade, Maven y JUnit para la elaboración de la estructura de pruebas. Ambos subproyectos han sido soportados por las tecnologías de Visual Studio Code, zsh e IntelliJ.

A través de mejoras continuas e integraciones graduales, se han desarrollado las siguientes tareas:

Primero, estableces una infraestructura de integración continua un GitLab junto con Docker para establecer un primer pipeline, aprendiendo las bases del funcionamiento de los pipelines y sus despliegues.

Posteriormente, se incorporó Maven y Jade para la integración del sistema multi-agente actualizando el pipeline, comprendiendo la función de los agentes y su proceso de construcción, compilación y despliegue.

A continuación, se incorporó la estructura de pruebas. Esta infraestructura automatiza el proceso de compilación y ejecución de pruebas, incluyendo pruebas unitarias, pruebas ontológicas y pruebas de integración entre agentes.

Una vez que la estructura estuvo finalizada, se ha creado un documento detallado que describe el proceso de instalación de un runner y los recursos necesarios para la nueva infraestructura, como el archivo gitlab-ci.yml y la estructura de pruebas generada permitiendo un mejor mantenimiento a lo largo del tiempo.

Una vez analizados los resultados, se han observado posibles líneas de desarrollo y mejoras que otorgarían un mayor valor al proyecto:

La instalación de los runners en un servidor de la escuela, que permitiría una mayor eficiencia y mejor rendimiento al aumentar la automatización de la infraestructura e independencia.

Integración del despliegue del proyecto en el pipeline junto con la adaptación del pipeline a nuevas tecnologías como Gradle.

Aun así, este avance permite mejorar la calidad y la eficiencia del desarrollo de prácticas mediante la detección temprana de errores. Todo ello permite una gran mejora, simplificando el proceso de trabajo y mejorando la calidad, fiabilidad, comprensión y gestión tanto de la nueva infraestructura como del proceso de desarrollo de la práctica.

# Abstract

This project is based on the study, development and implementation of an initial infrastructure and testing structure for the World of Agent practice.

World of Agents is the second assignment of the Agent-Based Software Development course of the prestigious European Master in Software Engineering in which they want to create the artificial players, the agents, of the proposed videogame.

For the development of the final degree project, Gitlab CI/CD technology was used for the infrastructure, together with the creation of runners and Docker technology. On the other hand, Jade, Maven and JUnit technologies were used for the development of the test structure. Both subprojects have been supported by Visual Studio Code, zsh and IntelliJ technologies.

Through continuous improvements and gradual integrations, the following tasks have been developed:

First, we set up a continuous integration infrastructure in a GitLab together with Docker to establish a first pipeline, learning the basics of how pipelines and deployments work.

Subsequently, Maven and Jade were incorporated for multi-agent system integration by updating the pipeline, understanding the role of agents and their build, compilation and deployment process.

Then, the testing structure was incorporated. This infrastructure automates the process of test compilation and execution, including unit testing, ontology testing and inter-agent integration testing.

Once the infrastructure and test set were finalized, a detailed document has been created describing the process of installing a runner and the necessary resources for the new infrastructure, such as the gitlab-ci.yml file and the generated test framework allowing better maintenance over time.

Once the results have been analyzed, possible lines of development and improvements have been observed that would give greater value to the project:

Installation of the runners on a college server, which would allow greater efficiency and better performance by increasing infrastructure automation and independence.

Integration of the project deployment in the pipeline together with the adaptation of the pipeline to new technologies such as Gradle.

Even so, this advance allows improving the quality and efficiency of practice development by early detection of errors. All this allows a great improvement, simplifying the work process and improving the quality, reliability, understanding and management of both the new infrastructure and the practice development process.

# Tabla de contenidos

# 1 Introducción

## 1.1 Motivación

El proyecto tiene como objetivo desarrollar un sistema de integración y pruebas de agentes inteligentes a desplegar en una plataforma escalable basada en contenedores software.

El objetivo es la ayuda a los estudiantes en la práctica de la asignatura Agent Based Software Development del Máster Universitario European Master in Software Engineering. Debido a que el trabajo está diseñado para que interactúen los subsistemas entre distintos grupos de alumnos, es necesaria la creación de una primera infraestructura DevOps que permita la validación automática de agentes software donde comprobar su correcto funcionamiento y su integración con el de otro grupo, permitiendo la independencia entre los subsistemas (plataforma y servicio).

La validación consiste en automatización de las pruebas de integración entre los agentes software que interconectan ambos subsistemas con una interfaz común y el desarrollo de una estructura de pruebas unitarias, para mejorar calidad del resultado final, utilizando las tecnologías de Gitlab CI/CD y Docker.

Este proyecto surge como una respuesta a la necesidad identificada en el ámbito educativo de mejorar el proceso de compilación e integración entre los alumnos, con el fin de optimizar su rendimiento académico y minimizar la pérdida de tiempo en actividades secundarias. El objetivo principal es permitir que los alumnos se enfoquen de manera más efectiva en el contenido de la asignatura y alcancen los objetivos de aprendizaje de manera más eficiente.

## 1.2 Enfoque y Planificación del trabajo

Para la elaboración del trabajo de fin de grado, se debía en tener en cuenta la variedad de tecnologías utilizadas y su sincronización entre ellas para el desarrollo del sistema multi-agente. Como se podía identificar tres grandes pilares en el proyecto, se optó el clásico *"Divide y vencerás"* en el cual, al principio del proyecto se crearían tres iteraciones, separando infraestructura, sistema multi-agente y estructura de pruebas que posteriormente irían incorporándose. Se realizaron los siguientes pasos:

1. Comenzando con Gitlab CI/CD, tanto runner cómo pipeline, y comprendiendo su funcionamiento junto a Docker, desarrollar una primera versión para afianzar el aprendizaje de estas nuevas tecnologías.
2. Incorporación de Jade y herramientas de compilación, que como veremos más adelante elegimos Maven, para desarrollar una segunda versión incorporando a los agentes en su instalación, compilación y extracción del fichero compilado .jar.
3. Incorporación de la estructura de test junto con JUnit para las funciones de los agentes, su ontología y su integración entre ellos. Además, desarrollo de un documento para su utilización y mantenimiento.

En cada uno de los procesos fue necesario la investigación, diseño, implementación, pruebas y mejoras para alcanzar el resultado final.

Para que estos objetivos se llevasen a cabo, se determinaron reuniones cada dos semanas para observar el progreso, analizar posibilidades y llegar acuerdos a la hora de desarrollar las pruebas. Estas reuniones, también servían para ayudar en caso de bloqueos o reconducir enfoques en el objetivo final y sus pasos intermedios.

## 1.3 Estructura del documento

El presente documento, contiene las siguientes secciones para facilitar su lectura:

**2. Estado de las tecnologías.** Información de las tecnologías utilizadas así cómo su propósito en el proyecto. Por parte de la infraestructura, estructura de test, la tecnología del sistema multi-agente e información relevante para su desarrollo cómo todas las tecnologías complementarias para llevarlo a cabo.

**3. Desarrollo.** Explicación de los pasos seguidos a la hora de la investigación y desarrollo del proyecto, así como cada una de las versiones que se fueron alcanzando para llegar al resultado final. En los últimos puntos se muestran los resultados obtenidos.

**4. Resultados y conclusiones.**

**5. Análisis de Impacto.**

**6. Bibliografía.**

**7. Anexos:**

**Anexo 1.** Código e imágenes de la primera versión del pipeline

**Anexo 2.** Código e imágenes de la segunda versión del pipeline

**Anexo 3.** Código e imágenes de la tercera versión del pipeline

**Anexo 4.** Fichero de código gitlab-ci.yml cuarta versión

**Anexo 5.** Fichero de código .iml versión IntelliJ

**Anexo 6.** Manual y documentación

# 2 Estado de las tecnologías

## 2.1 Gitlab

GitLab es una plataforma de DevOps (*Development and Operations*) en el que provee repositorios privados y abiertos, con una amplia gama de herramientas que nos da soporte durante el ciclo de vida completo del desarrollo de software, entre las más conocidas la gestión de proyectos y el control de versiones [1], [2].

Dentro de las herramientas que tiene GitLab, tiene un conjunto de ellas de integración y entrega continuas (CI/CD), conocido como GitLab CI/CD. CI/CD tiene dos partes.

CI es la práctica de integrar los cambios realizados en las ramas de desarrollo con el código principal y automáticamente probar los cambios actuales y previos para detectar errores o mal funcionamientos del código en una etapa temprana del desarrollo. Durante el proceso, se encarga de lanzar una compilación, permitiendo al desarrollador centrarse en su tarea y seguir avanzando sin tener que estar desplegando una vez consolidado un cambio. A través de ir consolidando los cambios frecuente mente, permite detectar conflictos en el códigos y errores de manera prematura, por lo que minimiza el impacto en el sistema. [3]

CD funciona en conjunto con la parte de CI. Una vez comprobado, testeado y compilado, CD se encarga de ver que tiene lo necesario, paquetes, librerías y otras dependencias para desplegar el sistema.

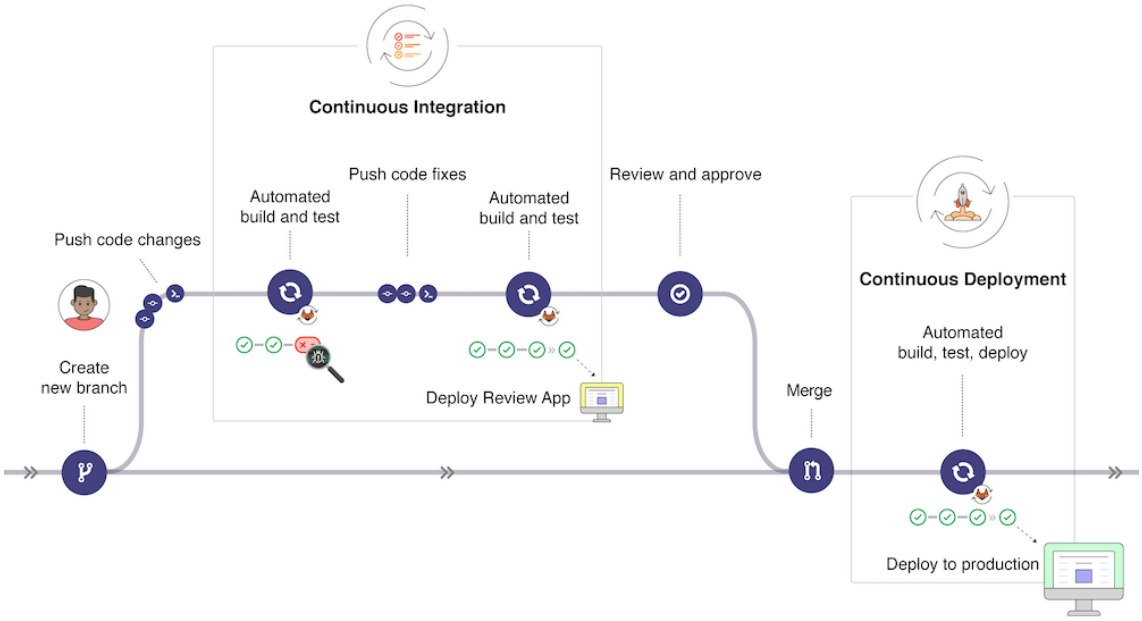



**Fig. 1** Flujo de un pipeline CI-CD [4]

GitLab CI/CD trabaja a través de sus pipelines, que son procesos automáticos en los que se comprueba el código para su integración y despliegue. Para poder realizar el flujo son indispensables los runners. Un runner es un agente que realiza las tareas del pipeline y envía de vuelta al repositorio los resultados [4]. Es una pieza clave en el proceso de creación de los pipelines, ya que junto al

fichero yaml, son los encargados de la realización de toda la parte de CI. Este fichero define las etapas, las dependencias y cuándo y cómo se ejecutarán los trabajos del pipeline. Tiene distintas etiquetas que realizan diversas funciones, aunque las principales son: *image*, *stages* y *script*. En *image*, se especifica el lenguaje o *frameworks* que necesita el script para poder funcionar. En *stages* se definen los pasos que va a recorrer el pipeline. Por último, en script se inserta el código que va a ejecutar el pipeline.

En este proyecto, al ser para una asignatura en la que van desarrollando varios equipos y se ponen de acuerdo en el funcionamiento del sistema, se necesita un entorno donde se pueda probar tanto la integración, cómo las ontologías y los métodos que acuerden entre ellos. Para que cuando lleguen a las reuniones, sólo tengan que desplegar el sistema y no inviertan esfuerzos y recursos en unas comprobaciones que pueden realizarse automáticamente.

Por eso Gitlab y sobre todo la parte de CI/CD es una parte fundamental de este proyecto, ya que, con sus herramientas, brindaremos al proyecto de un sistema de integración continua, en el cual se realizarán las comprobaciones y las pruebas antes de su despliegue, asegurando un mínimo de calidad en el sistema.

## 2.2 Docker

Docker es una plataforma de contenedores compactos, que fue diseñada para simplificar la creación, desarrollo y ejecución usando contenedores [5]. Permite la ejecución en un entorno virtual a partir de los componentes, paquetes y librerías necesarios para su despliegue. Y lo más importante, asegura que con cualquiera que compartas el entorno, funcionará de la misma manera, impidiendo errores asociados a la compatibilidad del ordenador o sistemas operativos [6].

Docker utiliza una arquitectura cliente-servidor, donde el cliente docker se comunica con el deamon, que se encarga de ejecutar, construir y distribuir el contenedor como vemos en la Figura 1.

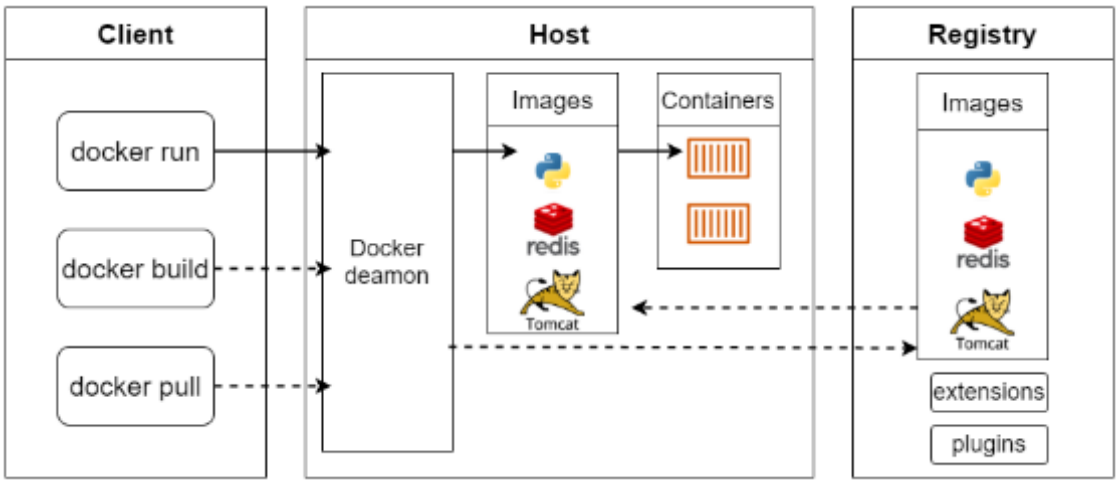



**Fig. 2** Ejemplo de arquitectura docker [7]

Para ejecutar los pipelines de GitLab CI/CD, el runner necesita un entorno para realizar las compilaciones y comprobaciones, por eso, Docker ofrece un entorno seguro en el que puede levantar una imagen que es ideal para que el runner ejecute los scripts de manera que no afecte ni se vea afectado por el usuario. Además, es compatible con Maven y el resto de las tecnologías, por lo que ayuda a su vez a la integración del sistema.

Por último, debido a que el proyecto de esta asignatura va creciendo a lo largo del curso y los alumnos requerirán distintos despliegues y pruebas a lo largo de las semanas, la eficiencia de Docker al permitir una creación y destrucción rápida de los entornos agiliza el proceso de desarrollo sin que los alumnos tengan que actualizarlo constantemente. Además, su escalabilidad permite que el proyecto vaya creciendo de manera segura, sin necesidad de ampliar constantemente el espacio de almacenamiento.

## 2.3 Maven

Maven es una herramienta de compilación para proyectos Java que ofrece varias ventajas. La principal de ellas es que automatiza el proceso de compilación, prueba, empaquetado y despliegue de los productos, asegurando una reproducción sencilla y menos errores [8]. Además, cuenta con un sistema de gestión de dependencias que facilita la construcción al permitir indicar el uso de otras librerías o proyectos necesarios, encargándose de su vinculación durante el montaje. Por último, estas dependencias pueden estar almacenadas tanto a nivel local como en repositorios públicos o en Internet, desde donde se extrae todo lo necesario para el funcionamiento del proyecto en el momento de la instalación.

Para realizar la configuración necesaria, se realiza a través del archivo pom.xml, en el cual se definen todos los aspectos relevantes, tales como la estructura base del proyecto, las dependencias requeridas, modificaciones en la configuración predeterminada de la construcción, así como las bibliotecas y complementos necesarios para dicha tarea. Este archivo es altamente personalizable, lo que facilita su adaptación a las necesidades particulares. Asimismo, goza de una amplia difusión y existen numerosas referencias disponibles para orientar y optimizar el proceso de configuración.

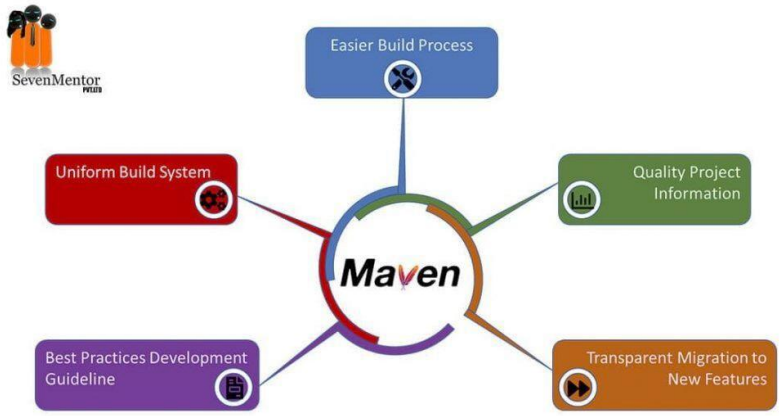



**Fig.3** Algunas ventajas de Maven [9]

## 2.4 Gradle

Otra de las herramientas más utilizada para la compilación de Java es Gradle. Tiene buenas ventajas como su flexibilidad y adaptabilidad al igual que un gran rendimiento, donde en proyectos grandes y complejos reluce especialmente [10].

Debido a las características del proyecto, que no sería de grandes dimensiones y que su configuración se quería dejar definida sin necesidad de modificación por parte de los alumnos, se eligió Maven debido a su mayor facilidad de uso, permitiendo que las pruebas en local sean más agiles, así como por al número de usuarios que facilita la solución a problemas comunes y la obtención de recursos.

## 2.5 Junit

JUnit es un framework de pruebas unitarias para la plataforma Java. Permite escribir y ejecutar pruebas automatizadas para verificar el correcto funcionamiento de componentes individuales o conjuntos de un proyecto. [11]

En un proyecto Maven, JUnit se utiliza como una de las dependencias a través de junit-jupiter dependency del proyecto para poder escribir y ejecutar pruebas unitarias de forma automatizada. Esto permite llevar a cabo pruebas de regresión de manera eficiente, asegurando que los cambios realizados en el código no introducen nuevos errores en el sistema. [12]

Además, JUnit permite integrarse con herramientas de construcción y despliegue continuo como GitLab CI/CD, facilitando la automatización del proceso de pruebas en un proyecto Maven. Esto resulta en una mejora en la calidad del software al detectar errores de forma temprana en el ciclo de desarrollo.

Una de las grandes ventajas de JUnit es su sencillez, ya que con unas cortas anotaciones se integran las pruebas al repositorio, por lo que hace que los alumnos de la asignatura puedan ir agregándolos a medida que van desarrollando las funciones para los agentes. A parte de las pruebas unitarias, en este proyecto se ha utilizado para filtrar y normalizar el formato de las ontologías.

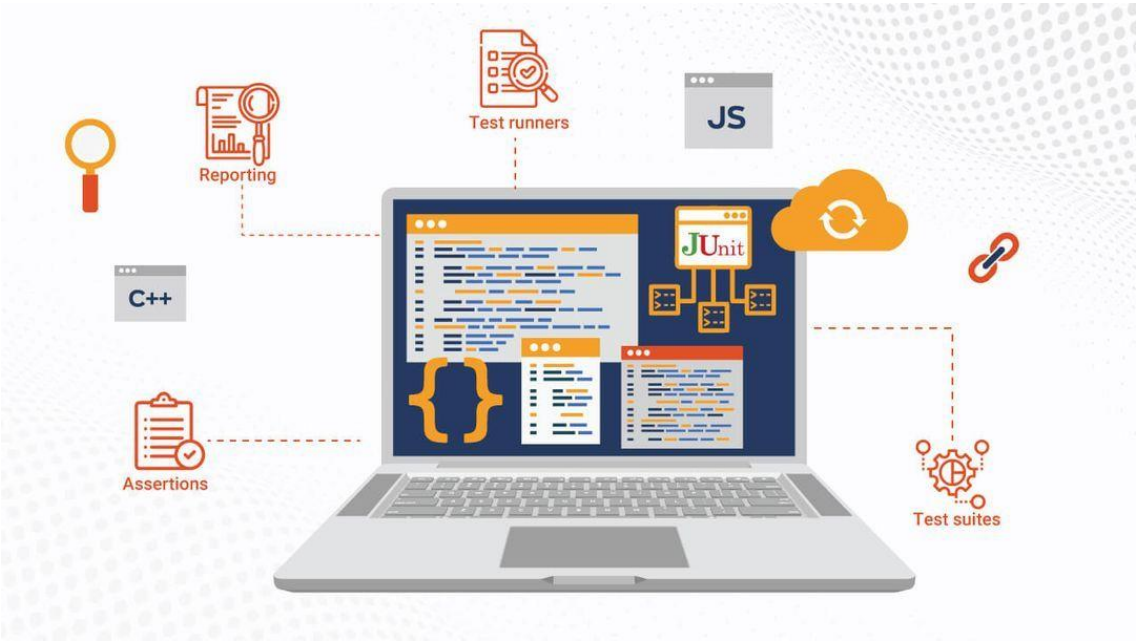



**Fig. 4** JUnit facilita el diseño de pruebas a través de etiquetas y generando reportes intuitivos

## 2.6 Jade

Jade, también conocido como Java Agent Development Framework, es un software utilizado para desarrollar aplicaciones basadas en agentes en sistemas multiagentes [13]. Este framework es fundamental en la práctica de la asignatura, donde los estudiantes deben implementar sus propios agentes Jade. Está compuesto de un entorno de ejecución, unas librerías para la implementación de los agentes y herramientas gráficas para su gestión.

El entorno de Jade crea un contenedor donde puedan habitar los agentes y relacionarse entre ellos. Para poder gestionarla, se genera un contenedor donde se despliegan el RMA (remote management system) para la gestión del sistema, AMS (agent management system) que otorga un nombre único a cada agente y el DF (Directory facililtator) que permite ver descripciones de los agentes registrados. Para poder levantar este entorno, dentro del Main hay que inicializarlo con la llamada java jade.Boot [opciones] [Lista de Agentes Iniciales] en la cual crea el contenedor y despliega un entorno junto con los agentes que hayamos asignado en la lista, aunque posteriormente también es posible la creación de más agentes y su registro.

Jade esta basado en el estándar FIPA (Foundation for Intelligent Physical Agents) que promueve el desarrollo y la estandarización de tecnologías para sistemas multiagente [14]. Estos agentes se basan en comportamientos, previamente programados y que se le asignan a un agente concreto a través de la llamada addBehaviour(). Los comportamientos se refieren a cómo interactúan y se relacionan los agentes entre sí dentro del sistema, como se puede observar en la Figura 5.

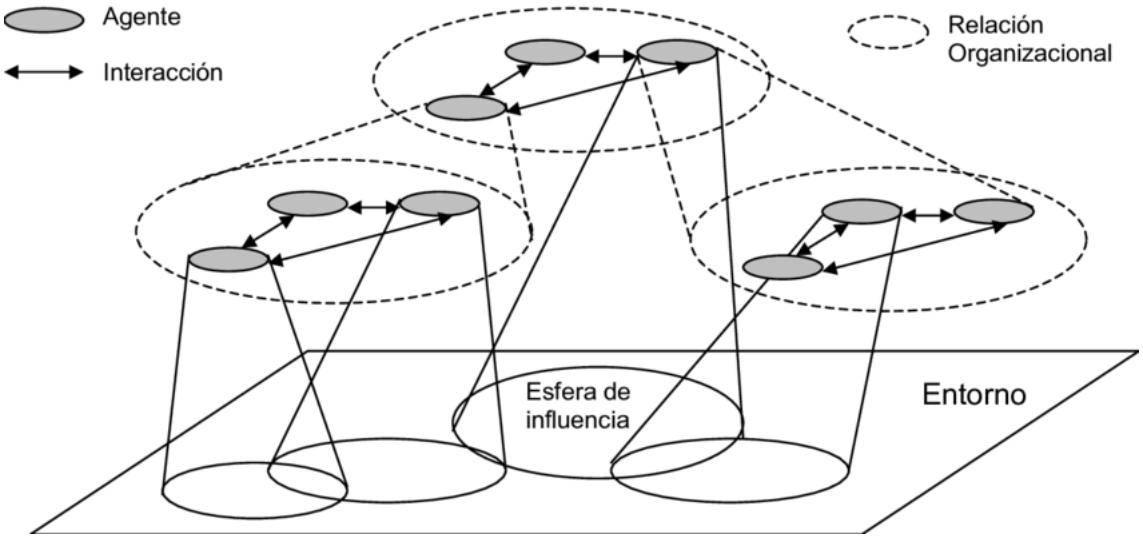



**Fig. 5** Representación de un sistema multiagente [15]

Al ser un framework de Java, es compatible con Maven y el resto de las tecnologías citadas anteriormente. Maven se encargará de su compilación, JUnit facilitará la creación de la estructura de pruebas, Docker creará un contenedor donde ejecutarse las pruebas que realizará el runner de GitLab siguiendo el fichero gitlab-ci.yml y harán que pasen o no el pipeline.

Por último, Jade a través del estándar FIPA (Foundation for Intelligent Physical Agents) cuenta con cierto grado de acuerdos a la hora de la comunicación entre los agentes, pero es necesario un vocabulario y una semántica concreta para llegar a comunicarse entre ellos [34]. La ontología es una representación formal y explícita de conceptos en un dominio específico, así como de sus propiedades y de las relaciones entre ellos. En otras palabras, la ontología en sistemas multi-agente define el vocabulario y la semántica junto con su estructura para el contenido de los mensajes entre los agentes. En Jade, se implementa extendiendo de la clase Ontology y añadiendo en el sistema un conjunto de conceptos, acciones y predicados que compondrán el contenido de los mensajes. Para desarrollar la ontología, tenemos tres interfaces: Concept, AgentAction y Predicate. Las clases correspondientes para esas interfaces serían: ConceptSchema, AgentActionSchema y PredicateSchema [34]. La jerarquía dentro de estas clases viene heredada por dos vías como observamos en la Figura 6, que al final nacen en la superclase ObjectSchema.

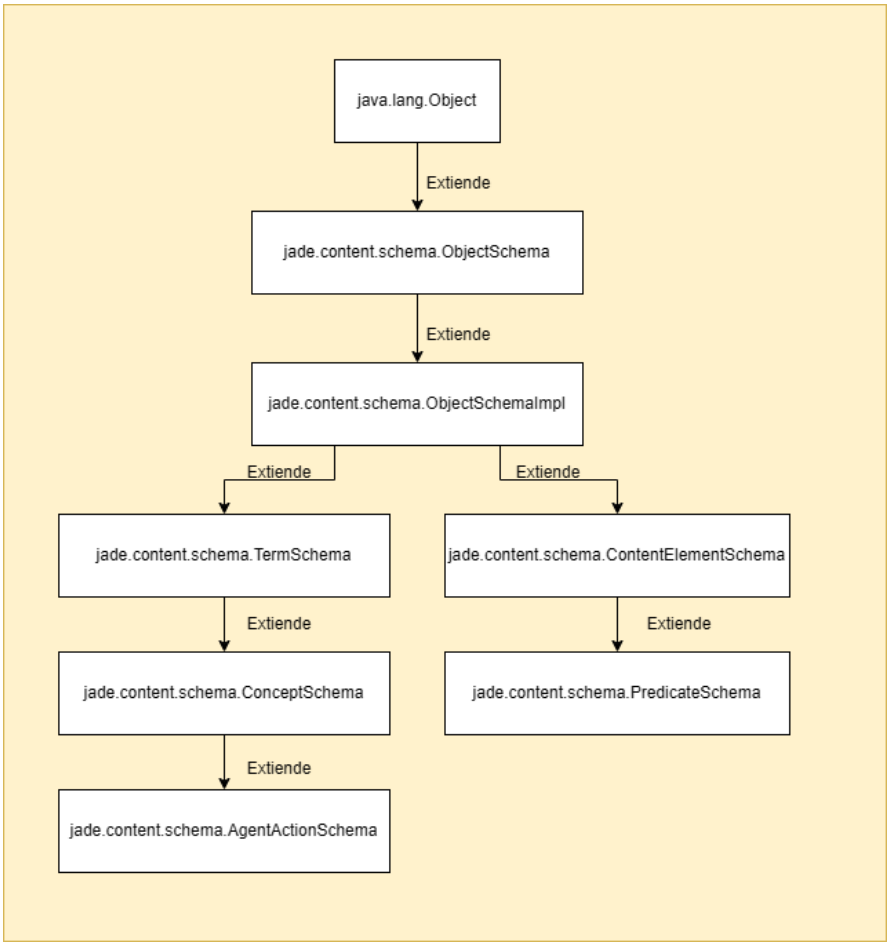



**Fig. 14** jerarquía de clases *Schema* de la ontología

## 2.7 Visual Studio Code

Visual Studio Code es un editor de código fuente desarrollado por Microsoft, disponible en las tres plataformas principales: MacOS, Windows y Linux [16]. Este editor, cuenta con multitud de funciones que facilitan tanto la escritura cómo la edición y depuración del código. Además, cuenta con una amplia variedad de extensiones que cubren tanto Java, como las herramientas de compilación Gradle y Maven, cómo el código del maven y su formato. Otra de las ventajas con la que cuenta este editor de texto, es que es posible realizar la clonación de gitlab a través del editor y posteriormente gestionar los comandos para subir cambios o descargarte versiones del proyecto vinculado.


**Fig.6** Extracto de código escrito en visual studio code.

## 2.8 Zsh

Zsh es un intérprete de órdenes o Shell, diseñada para su uso interactivo y eficaz lenguaje de scripting [17]. Contiene una gran variedad de características que ayudan tanto a la visualización del terminal, como un sistema de historial, autocompletado o generación de comandos acortados.

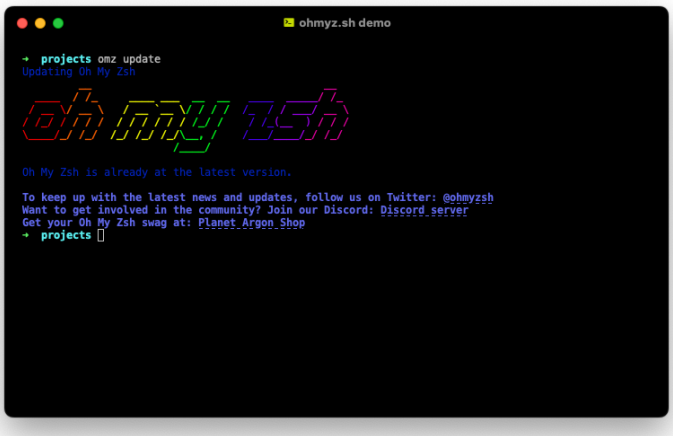



**Fig.7** pantalla de inicio del terminal cuando inicializas con la extensión oh my zsh, framework que maneja zsh [18]

## 2.9 IntelliJ

IntelliJ es un entorno de desarrollo integrado para el desarrollo de programas informáticos [19]. Tiene gran cantidad de herramientas avanzadas de desarrollo, depuración, escritura, refactorización e integración de diversas tecnologías, entre ellas Java, con gran apoyo en la comunidad de desarrollo Software.

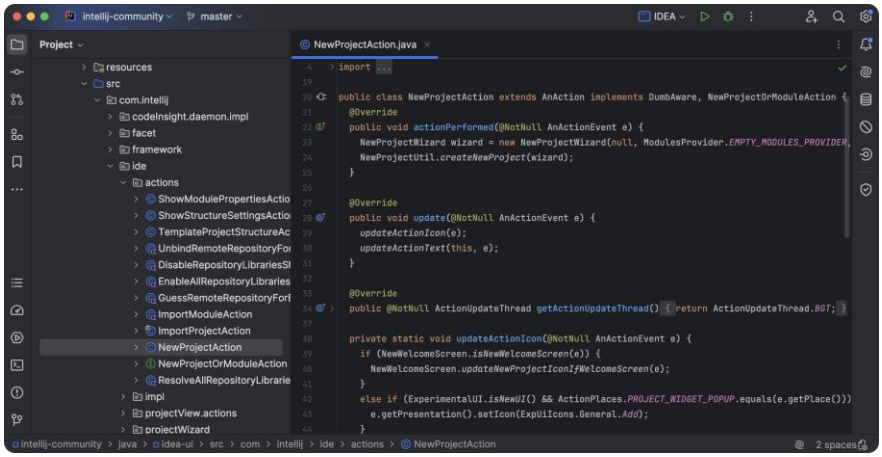



**Fig.8** Imagen de ejemplo del editor de IntelliJ [20]

# 3 Desarrollo

Para llevar este proyecto, se ha tenido que ir investigando las tecnologías previamente mencionadas e ir poco a poco incorporándolas en el proyecto hasta alcanzar el desarrollo del sistema de integración, como observamos en la siguiente figura.

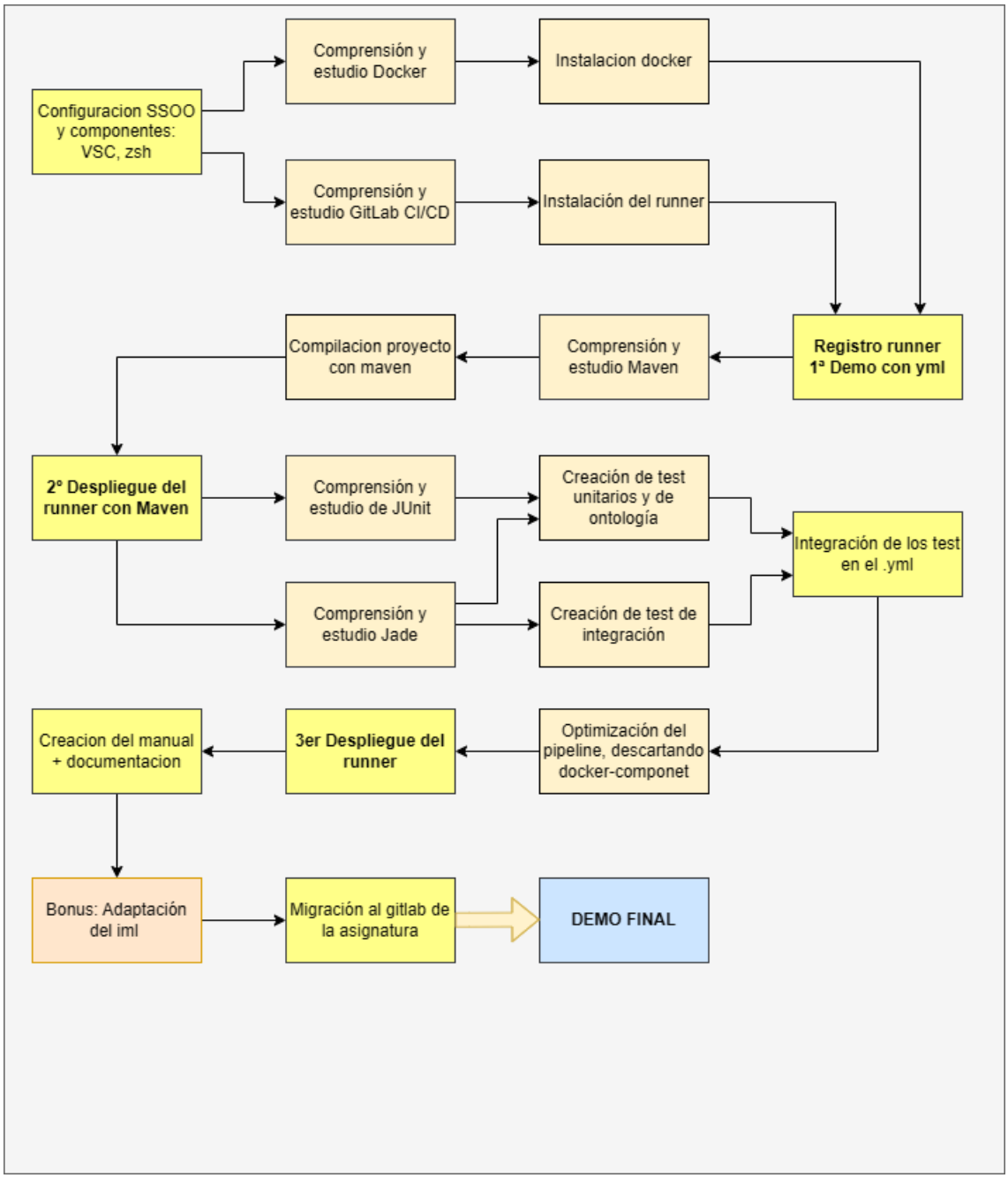



**Fig.9** Líneas de trabajo del proyecto

## 3.1 Trabajo previo

### 3.1.1 Contexto del proyecto y necesidades. World of Agents

El proyecto en el que se busca realizar la mejora es la práctica titulada "World of Agents". En ella, cada equipo debe crear un agente capaz de registrarse y llevar a cabo todas las acciones del juego para poder ganar al final de la partida. Los estudiantes deben ser capaces de construir edificios, mover unidades por el mapa, extraer recursos y crear nuevas unidades. Al final de la partida, a través de un cálculo de las puntuaciones obtenidas por cada objetivo u objetivos logrados se obtiene el ganador.

Actualmente, los alumnos tienen un Manual técnico para el uso de los repositorios, para la configuración de las versiones de Maven o de IntelliJ y el comando para la ejecución del fichero .jar y la estructura del esqueleto.

Debido a que el proceso de compilado y la generación de una estructura de pruebas se realizaban de manera manual y por cuenta propia de los equipos de alumnos, disminuía la calidad de los resultados y entorpecía a la hora de la integración entre distintos equipos. Por otro lado, los fallos manuales de compilación y errores de ejecución consumían gran tiempo de la clase que hacían que no pudiesen invertirse en un mayor aprendizaje.

Por todo ello, este trabajo de fin de grado tiene dos objetivos:

- Facilitarles la tarea de compilación e integración proporcionando una infraestructura que compile y ejecute las pruebas cada vez que consoliden el código en su rama.
- Generación estructura de pruebas que garanticen la calidad y les permitan centrarse en el desarrollo del agente, ayudando a la localización de errores de manera temprana en el desarrollo.

### 3.1.2 Preparación del entorno

Para la realización del proyecto, se optó por la instalación de Ubuntu 22.4 [21] debido a su rapidez de instalación, intuitividad y sus recursos adecuados para la programación. Se decidió instalarlo en un disco duro extraíble para facilitar su portabilidad y permitir una mayor capacidad de almacenamiento. Se eligió esta opción, en lugar de una máquina virtual, para aprovechar al máximo el hardware disponible y garantizar un rendimiento óptimo, así como simplificar la instalación de herramientas y la gestión de permisos.

Una vez completada la instalación, se procedió a configurar zsh, un Shell con plug-ins que mejoran la experiencia de trabajar en la terminal [22]. Además, para el desarrollo del proyecto se seleccionó Visual Studio Code por su amplia gama de extensiones y la experiencia previa positiva con este editor de código, que permite una mayor fluidez en el manejo del código.

Después de investigar, documentarse y aprender sobre las tecnologías necesarias para este proyecto, se empezó creando un proyecto en GitLab en donde se realizarían las pruebas dado que era necesario tener permisos de administración para trabajar con la infraestructura y las configuraciones de los pipelines. A su vez se comenzó instalando Docker Engine, que es el componente principal de las herramientas que componen Docker que nos permite crear, administrar y ejecutar los contenedores de la aplicación, así como el Daemon de docker [23]. Se realizó a través de la página de Docker, siguiendo los pasos de la sección de Ubuntu en el camino de Docker Desktop para Linux [24]. Una vez comprobado el correcto funcionamiento se comenzó con el desarrollo del proyecto.

El proyecto tiene dos partes fundamentales, la primera es la integración del sistema y generación de los pipelines y la segunda es la generación de una estructura de prueba para regresión y asegurar la calidad del proyecto.

## 3.2 Creación de la integración continua

La integración continua como se ha comentado en la sección de Gitlab CI/CD, tiene dos partes fundamentales:

- El fichero yml, que tiene las órdenes de ejecución del pipeline y todas las configuraciones necesarias para su funcionamiento.
- El runner, que es el encargado de la ejecución del pipeline a través de los scripts del yml.

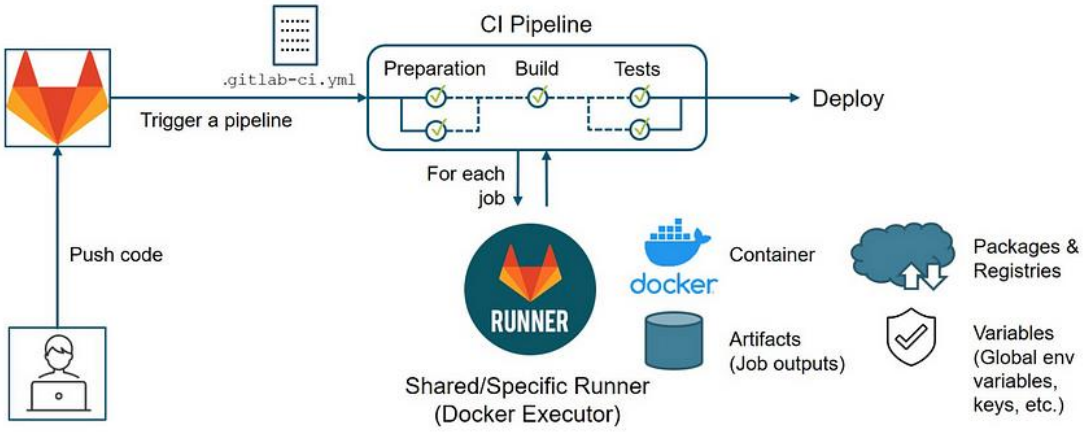



**Fig.10** Flujo de ejemplo de funcionamiento de Gitlab CI y ejecución del pipeline

### 3.2.1 Instalación del runner

Para registrar un runner de GitLab, lo primero que hay que hacer es instalarlo en nuestro entorno. Dentro de los posibles runners existentes, los primeros que descartamos eran los GitLab-hosted runners debido a que son de uso público y compartidos por la comunidad, por lo que ocasionaría retrasos en las ejecuciones. A su vez, son predeterminados por la plataforma Gitlab, por lo que no podríamos personalizarlos para nuestro proyecto. Por eso, elegimos self-managed runners a pesar de que utilizan los recursos de nuestro ordenador/servidor, debido a que son de ejecución privada y controlables por el usuario, lo que nos permite adaptarlos a nuestras necesidades y mejora los tiempos de ejecución, ya que no se comparten con otros usuarios de Gitlab [25]. Como nos permiten tener mayor control, podemos personalizar su configuración y los recursos que les otorgamos, ya que comparten nuestra misma infraestructura. Por último, al ser los más simples de instalación y registro, permiten a los alumnos crearse los suyos propios y así cada equipo tener su propio runner que ejecute los pipelines.

A la hora de instalación existen dos opciones, instalarlo en un contenedor o en el propio sistema GNU/Linux, MacOS, FreeBSD o Windows. Actualmente los runners soportan las siguientes arquitecturas: x86, AMD64, ARM64, ARM, s390x, ppc64le [26], compatible con que los alumnos puedan instalárselo en

sus ordenadores personales para tener su propio runner y abre las puertas a futuros desarrollos.

Al estar trabajando en un sistema operativo GNU/Linux lo instale por esa opción que se recoge posteriormente en el manual (3.6.1.1 Install GitLab runner (For Ubuntu)).

Una vez instalado en el entorno el runner, se empezó a hacer las primeras pruebas en el repositorio GitLab. Se creó una rama para seguir las buenas prácticas y solo subir a master versiones estables y probadas del desarrollo, siguiendo las buenas prácticas de gestión de proyectos [27]. Esta nueva rama se clonó en local para escribir el código, hacer las posibles pruebas en local y subir con cambios significativos a la rama.

Una vez creada la rama se procedió a registrar un primer runner. Al registrar un runner, le otorgas de unas características concretas y le asignas un tipo de trabajos que puede coger del GitLab. El registro se hace vía interfaz la primera parte, dentro del propio proyecto GitLab y una segunda parte vía terminal. Hablaremos de cada uno de sus campos y su importancia. En la sección de interfaz, el primero de todos es el campo Tags, que marca que etiquetas deben tener los Jobs para que este runner pueda correrlos. Por control de seguridad, es recomendable que todos los runners y steps en el yml tengan tags asociados, para impedir que se revele información sensible. Otro motivo por el que son necesarios los tags en este proyecto es debido a que como va a haber diversos equipos implementando sus propios agentes, evitar que el pipeline se quede atascado porque el runner de otro usuario coja ese trabajo y estén en distintas etapas del proyecto. La descripción del runner ayuda a identificar cual runner está ejecutando los trabajos, por lo que mejora la localización en caso de algún problema. La marca de paused se requiere cuando no quieres que acepte trabajos por el momento y la marca de protected, para que se ejecute en ramas protegidas, esto es útil en proyectos más grandes para reservar ciertos runners para hotfix o incidencias que quieres que tengan una prioridad mayor y sus propios recursos asociados. La marca de Lock to current Project es preferible marcarla por temas de seguridad. Y por último el maximun job timeout, es posible dejarlo en blanco, pero es recomendable rellenarlo para que si el pipeline se bloquea y no termina de avanzar sea el mismo runner el que cancele la ejecución y no se malgasten recursos. Se suele poner en un tiempo alto para que no genere falsos negativos. Todos estos campos una vez el runner se registra pueden modificarse, lo que facilita su corrección en caso de problema en la instalación.

Para la segunda parte, que se realiza en el terminal tiene menos campos, pero son cruciales para que el registro del runner sea correcto y pueda funcionar. Los primeros campos url y token, nos los proporciona gitlab para disminuir los errores de transcripción. En el nombre del runner es recomendable ponerle el mismo que en la descripción, para que en el config.toml se le identifique igual que en GitLab y no haya problemas de identificación. Por último, nos pide que elijamos un executor, el executor es el que nos permite ejecutar los scripts de CI en un escenario concreto y tiene diversas opciones: directamente en el Sistema Operativo (Shell, ssh), dentro de un contenedor (docker, docker-ssh), o en una máquina virtuaol (virtualbox, parallels). Este executor nos permite construir y probar en el entorno que seleccionemos [28]. En nuestro caso, seleccionamos docker, ya que ofrece una construcción limpia del entorno y todas sus dependencias se incluyen dentro de la Imagen de Docker, además si

ejecutamos varias veces seguidas sin cambiar el entorno, reutiliza la imagen aumentando la velocidad de ejecución de los pipelines aprovechando los recursos, como podemos ver en la Figura 3.

| Executor | SSH | Shell | VirtualBox | Parallels | Docker | Kubernetes | Custom |
|---|---|---|---|---|---|---|---|
| Clean build environment for every build | × | × | ✓ | ✓ | ✓ | ✓ | conditional (4) |
| Reuse previous clone if it exists | ✓ | ✓ | × | × | ✓ | × | conditional (4) |
| Runner file system access protected (5) | ✓ | × | ✓ | ✓ | ✓ | ✓ | conditional |
| Migrate runner machine | × | × | partial | partial | ✓ | ✓ | ✓ |
| Zero-configuration support for concurrent builds | × | × (1) | ✓ | ✓ | ✓ | ✓ | conditional (4) |
| Complicated build environments | × | × (2) | ✓ (3) | ✓ (3) | ✓ | ✓ | ✓ |
| Debugging build problems | easy | easy | hard | hard | medium | medium | medium |

**Fig. 11** opciones de executors y sus capacidades [29]

Al seleccionar el executor de docker nos pide que insertemos una imagen por defecto. Al principio se utilizó alpine:latest ya que sólo queríamos probar la ejecución del pipeline y el funcionamiento del runner, pero debido a que el proyecto que tienen que hacer los alumnos de la asignatura es sobre Sistemas Multi-agente utilizando JADE, la imagen final fue openjdk:17 ya que es la versión de java que se va a utilizar en el proyecto y es válida para cualquier herramienta de construcción (Maven o gradle), lo que nos otorga una mayor flexibilidad.

Por último, es importante destacar, que, a la hora de la instalación y registro del runner, es recomendable hacerlo en superusuario a causa de que algunas ejecuciones en los scripts, a la hora de instalar los paquetes y dependencias pueden verse afectados si no tiene los permisos suficientes. De la misma forma, no pueden acceder a lugares protegidos del sistema por lo que se limitarían en gran medida los recursos disponibles.

Una vez creado el runner, se añadieron las instrucciones en el README del proyecto para que pudieran consultarlo cuando lo necesiten.

### 3.2.2 Creación del yml

El fichero gitlab-ci.yml es el encargado de configurar y definir los distintos Jobs y flujos de trabajo que se ejecutarán como parte de la integración continua en el proyecto. Durante el desarrollo se fueron realizando distintas versiones hasta alcanzar el resultado final.

#### 3.2.2.1 Primera versión

La primera versión, era un fichero muy primitivo en el que se comprobaba el funcionamiento del runner y del pipeline. Como se puede observar en el Anexo 1, ese yml era solamente un hola mundo del pipeline.

#### 3.2.2.2 Segunda versión

Una vez comprobado el funcionamiento del pipeline, se estuvo investigando los distintos componentes y etiquetas que puede tener un yml. En el Anexo 2 se puede observar una de las comprobaciones que consistió en la opción de ejecutar en el mismo step distintos scripts, funciona poniendo distintos nombres a los *steps* pero con el mismo nombre en la etiqueta *stage*, por lo que se ejecutan de manera paralela, es bueno si quieres dividir un script de un step para reducir tiempos o reducir su complejidad.

#### 3.2.2.3 Tercera versión

Una vez finalizadas las pruebas y profundizado en el proyecto y sus tecnologías, se presentó un primer diseño del pipeline (figura 4) en el cual, se centraba en la parte de infraestructura y en levantar el proyecto en un contenedor Docker. Se separaban 4 partes fundamentales de una compilación, omitiendo la parte de Test y comprobaciones que se añadiría más adelante.

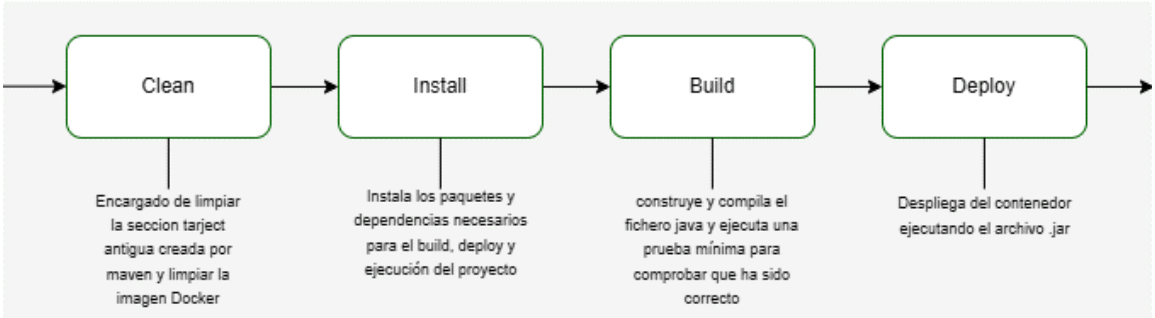



**Fig. 12** Diagrama de flujo de la versión 1 del pipeline

El primer step, llamado *Clean*, se encarga de preparar el entorno, eliminando cualquier resto de ejecuciones o despliegues pasados para evitar contaminaciones. A continuación, *Install*, realiza la instalación de dependencias y paquetes necesarios tanto para el proyecto, Maven y docker para que funcionase correctamente. En tercer lugar, *Build*, construye y compila el proyecto, comprobando al finalizar que ejecuta una orden mínima: `java -jar jade-woa-skeleton-maven-1.0-SNAPSHOT-jar-with-dependencies.jar`, que

devuelve información sobre el proyecto y los comandos disponibles. Y para finalizar, *Deploy*, ejecuta el montaje y despliegue en la imagen Docker a través del comando docker-compose. Este es un subcomando que se encarga de construir y gestionar los contenedores Docker a través de un fichero llamado docker-compose.yml [30] y un Dockerfile contiene instrucciones para levantar la nueva imagen [31]. El código del Dockerfile es sencillo, como se puede observar en el Anexo 3, simplemente se indica la imagen, el espacio de trabajo, que recursos son necesarios y la ejecución una vez levantada la imagen. Mientras que el fichero docker-compose.yml, establece que habrá dos servicios: el primero un servicio Front-End que se levantará en local en el puerto 8080 y sería la interfaz, aunque no se define por el momento los comandos, ya que aún no se había interactuado con la interfaz que tienen en la asignatura. Y el segundo, que sería el *backend* que sería el Sistema Multi-Agente del proyecto. En ese momento el archivo ejecuta el Dockerfile en web.

El yml tenía los cuatro pasos mostrados en la Figura 4 y ejecutaba el pipeline correctamente como se puede observar en el mismo anexo. En esta nueva versión, se añadieron los *stages* (etapas del pipeline) pensados para la posible integración final y se dio más forma a cada uno de sus scripts ya integrados con Maven:

- El *stage* de *clean*, a través de ese comando eliminaba los archivos compilados en tarjet en ejecuciones anterior.
- El *stage* de *install*, instala todas las dependencias y después con el comando *verify* comprueba que los test pasen correctamente, aunque en ese momento aún no se había integrado JUnit al proyecto.
- El *stage* de *build*, compila el proyecto jade y ejecuta el *help* para comprobar que funciona correctamente, a su vez, se le añadió la sección de *artifacts*, que permite descargar el fichero compilado en esa ejecución concreta. Esta mejora es un punto importante debido a que nos permite exportar entre los distintos equipos su .jar más reciente y en caso de errores, versiones de su .jar previo, pudiendo hacer una traza en caso de error. A parte, también permite que un equipo extraiga el .jar de otro equipo y pueda hacer pruebas con múltiples agentes sin tener que depender de se lo estén generando y enviando cada vez.
- El *stage* de *deploy*, es el único que utiliza una imagen docker, ya que es necesario para la ejecución de sus comandos. A parte, se le añade el service: docker:dind para que pueda trabajar dentro de la imagen del runner y generar la imagen del proyecto.

Para que el yml funcionase correctamente en esta versión, también se modificó el pom.xml, este archivo es el alma de la herramienta de compilación y es la que controla tanto las dependencias, las versiones y cómo se tiene que construir y levantar. Se decidió actualizar el *lombook*, una librería que da soporte a Java, a una nueva versión debido a la pérdida de soporte de la versión anterior.

Para realizar esta versión se obtuvo un esqueleto del proyecto del curso anterior con el que se fue trabajando, adaptando y mejorando para añadir las nuevas funcionalidades. De los dos esqueletos disponibles, se eligió el de Maven para empezar a trabajar con él.

### 3.2.2.4 Cuarta versión

Posterior a la demostración de la tercera versión, llevamos a cabo diversas reuniones en las que ajustamos el flujo y añadimos objetivos que tuvieron un impacto significativo en la sección de pruebas, generando repercusiones en el pipeline, dado que es el encargado de realizar la regresión.

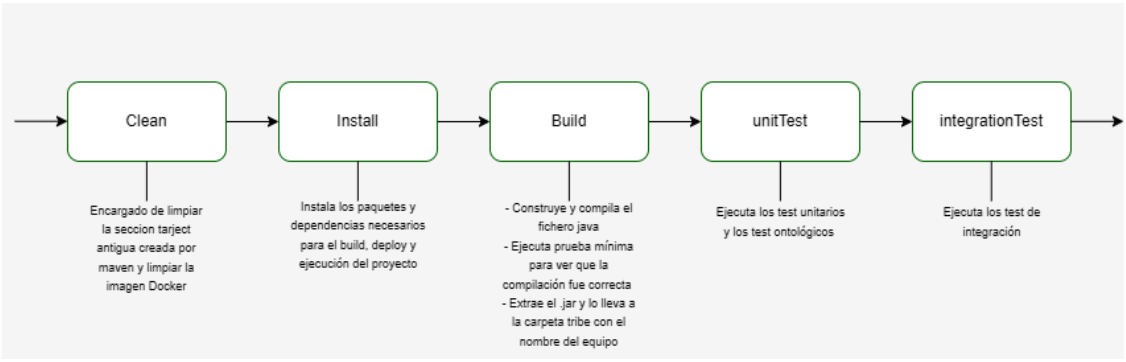



**Fig. 13** Diagrama de flujo de la versión 1 del pipeline

En la Figura 5 se puede apreciar que mantuvimos los dos primeros pasos sin cambios, pero se completó el paso del *Build*, puliendo los últimos detalles. Además, se agregaron dos secciones de pruebas: una para pruebas unitarias y ontológicas, facilitando su ejecución conjunta a través de Maven y JUnit, y otra para pruebas de integración. Se decidió no incluir el despliegue del entorno, ya que la ejecución con la interfaz se haría solo al final del semestre, lo cual aumentaría significativamente el costo en los pipelines. Se optó por mejorar la velocidad de ejecución e implementar la integración junto con la regresión, así como crear un manual para que los alumnos pudieran crear sus propios runners, reduciendo así el esfuerzo necesario para la integración y permitiéndoles enfocarse más en su proyecto y en la creación de sus agentes. Una vez que hubieran desarrollado una parte, podrían ejecutar el pipeline para verificar que funciona correctamente y obtener el archivo .jar compilado.

Todo esto hizo que el yaml cambiara significativamente como observamos en el Anexo 4. Lo primero de todo se colocó al comienzo del script las imágenes tanto de Maven cómo de docker, ya que ningún paso necesitaba una imagen distinta al resto. A través de la imagen docker:latest, se consigue que se ejecute el pipeline en un entorno dockerizado, lo que simplifica el código y evita complicaciones con el despliegue a través del docker-compose. Para completar la imagen, se selecciona Maven:3.8.8 para especificar la versión que va a ejecutar el pipeline, haciendo que se ejecute en las mismas condiciones que harán los alumnos en local y evitando así problemas de compatibilidad.

Los *stages* resultantes son: *clean*, *install*, *build*, unitTest e integrationTest.

- **Clean**: se mantuvo igual que en la versión anterior, ya que solo es necesario eliminar el tarjet previo si existiese.
- **Install**: dejamos ese step para la instalación de las dependencias y paquetes necesarios y se eliminó la verificación del proyecto.

- **Build**: Se modificó el flujo de tal manera que, una vez compilado el proyecto, se le asigna el nombre woa-{nombre_del_equipo}.jar y se traslada a la carpeta tribes, en esta carpeta, se almacenará todos los agentes de los equipos para que el último realice el despliegue del entorno llamando a esos agentes. Una vez movido a esta carpeta, se ejecuta el comando de *help*, para comprobar que funciona correctamente la compilación y no hay errores que impidan su uso.
- **unitTest**: realiza los test unitarios y ontológicos, como sus verificaciones. Hablaremos en más profundidad en la siguiente sección. Debido a que los test se pueden ir incrementando y a la hora de la reunión de los equipos puede no haberlos completado, se le otorgó a este paso la capacidad de fallar (allow_failure:true) sin que pare el pipeline, para poder verificar que la ejecución del test de integración es correcta. Igualmente, en el pipeline aparece una exclamación "!" si ese paso falla.
- **integrationTest**: extrae el nombre del equipo y ejecuta los test de integración.

Debido a que se descartó el despliegue, se eliminó el fichero docker-compose ya que no era necesario para la ejecución, al añadir image:docker se genera el contenedor docker donde el runner ejecuta las pruebas y una vez realizadas se elimina, disminuyendo los recursos y la carga del sistema local que soporta el runner.

Una vez generado el pipeline final y comprobado su funcionamiento, se creó la primera parte del manual, en la que se recoge la generación del runner e información básica sobre el fichero yml.

## 3.3 Creación de los test

La práctica sobre la que se desarrolla esta infraestructura y pruebas tiene una característica que lo convierte en un reto. Los equipos, formados por grupos de alumnos que cursan la asignatura, tienen que elegir una semántica común para que los agentes Tribu se comuniquen con el agente Plataforma e interactúen entre ellos. Es decir, la definición de los métodos y desarrollos de comunicación de los agentes se deciden mediante acuerdos y negociaciones por parte de los equipos. Por otro lado, una vez acordado el envío y respuesta de información, cada equipo realizará de forma independiente el funcionamiento de los métodos de su propio agente, utilizando las funciones auxiliares que consideren necesarias. Esto hace que la generación de pruebas de regresión no sea posible, debido a que los propios métodos no están definidos, pero es posible establecer una estructura para que les sea más fácil implementar las pruebas una vez los acuerdos se hayan realizado.

Para realizar los test unitarios y ontológicos, se incorpora JUnit como recurso de manejo de test. Por ello, en el pom.xml que controla Maven, se han añadido la versión jupiter 5.8.1 de JUnit con la que se va a realizar los test, y maven-surefire-plugin para la configuración y ejecución de estas.

En cada apartado hablaremos del enfoque para cada conjunto de pruebas realizadas para el proyecto.

### 3.3.1 Test Unitarios

Los test unitarios son pruebas que se realizan a nivel de un componente, función, módulo o método específico dentro de un programa y se ejecuta según lo esperado por el desarrollador [32]. Normalmente, para la generación de test unitarios se estructura en tres tipos:

1. *Happy path*: resultados que esperan que sean correctos.
2. *Limit path*: resultados dentro de los límites de la función, tanto correctos como incorrectos.
3. *Unhappy path*: resultados incorrectos que esperas determinado error o fallo.

Las pruebas unitarias validan y ofrecen calidad al código escrito por el desarrollador, evitando fallos en etapas tempranas del desarrollo. Las pruebas deben ser legibles y mantenibles en el tiempo, ya sea por el desarrollador u otros miembros de su equipo [33]. Además, cada caso de prueba tiene que ser un caso aislado, del que no le influyan el resto de los componentes para poder probar su correcto funcionamiento.

Como hablábamos en el apartado anterior, debido a las condiciones de esta asignatura, para los test unitarios se ha generado una estructura de clases para facilitarles la generación y ampliación de los test.

Se generaron clases espejo, en las cuales se ejecutarán los test unitarios de cada clase. Para la clase Main.java, se creó la clase MainTest.java, para la clase Utils.java se creó la clase UtilTest.java, etc.

Para la clase UtilTest.java se ha creado 4 primeras pruebas unitarias para las funciones ya existentes y unas configuraciones previas a la generación de los test:

- testNewMsg(): comprueba que se ha almacenado el contenido y el performative esperado y que el remitente sea null.
- testSimpleNewMsg(): comprueba que el performative sea el esperado cuando se llama con dos parámetros a la función.
- testRegisterVoid(): comprueba que un nuevo agente no esté registrado aún.
- testGENCID(): comprueba que genera un CID a un nuevo agente correctamente.

La clase AgentTest.java está preparada para que se inserten las pruebas unitarias del agente Tribu que genere el equipo concreto, como al inicio la propia clase solo tiene una llamada a su comportamiento para generarse, se añadió un comentario documentando los primeros pasos para la definición de los test y algunas etiquetas útiles de JUnit como el @BeforeEach y un test básico de prueba.

La clase AnswerBehaviourTest.java llama se ha creado una prueba para la función existente en AnswerBehaviour.java a parte de su constructor:

- testSalute(): comprueba que se genera un mensaje de saludo correctamente.

Por último, la clase MainTest.java tiene tres test creados un poco más complejos que los anteriores y una función que, al finalizar cada uno, cierra el agente contenedor y apaga el runtime. Los test son los siguientes:

- testLoadPlatformAgent(): comprueba que se genera un agente Plataforma correctamente generando la estructura necesaria para su funcionamiento. Si se ha podido crear, dará true y se cerrará de nuevo, en caso de error devolverá un false.
- testLoadTribeAgente(): genera un agente tribu0 y comprueba que su inicio ha sido correcto, una vez comprobado se mata al agente y devuelve true.
- testLoadTribe{N}Agent(): genera un agente tribuN, donde N hay que especificar al equipo que pertenece y comprueba si su creación ha sido correcta. Este test se ha dejado comentado para que lo añadan en el momento de integración de agentes.

Estos 3 test para evitar problemas con la interfaz rma de los agentes, se ejecuta con el comando -nomtp que no genera la interfaz y evita problemas en la automatización.

### 3.3.2 Test de la ontología

En la asignatura "Agent-Based Software Development", los alumnos como comentábamos en los test unitarios deberán definir la ontología juntos, aún con más motivo que los propios métodos, debido a que es la que permite la comunicación entre los agentes. La ontología, como explicábamos en la sección de 2.6 Jade, define tres interfaces (AgentAction, Concept y Predicate) junto con un vocabulario para desarrollar la estructura conceptual. Para probar la ontología, se ha desarrollado una serie de test reflexivos para las interfaces y el vocabulario. Los test reflexivos son un tipo de prueba que se autoincrementan a medida que avanza el desarrollo. Suelen utilizarse para probar estructuras que no están definidas previamente. En las ontologías, el vocabulario se suele escribir en la clase *Ontology* que es la que define la estructura de la ontología utilizando los conceptos, acciones de agente y predicados declarados en sus propias clases.

En este caso, se han realizado 5 pruebas, intentando mantener la mayor flexibilidad posible para que los alumnos generen la estructura deseada, pero manteniendo los estándares definidos para las ontologías. En estas pruebas se comprueban cada una de las partes de la ontología. La intención de estos test es que, una vez llegado a un acuerdo, pruebe que todos los equipos utilizan la misma métrica ontológica, revisando que el *Vocabulary*, el contenido del *Vocabulary* y las estructuras de *Concept*, *AgentAction* y *Predicate* están como se ha puesto en los acuerdos de los alumnos. La base general de estos test se basa en que en la carpeta src.test.resources se encuentran cuatro ficheros, cada uno para cada estructura: *Vocabulary*, *Action*, *Predicate* y *Concept*. En el que se lista el contenido acordado en cada una de ellas y a partir de ahí comprueban en el sistema que está bien definido. Las pruebas son las siguientes:

- **vocabularyTest()**: extrae del fichero vocabularyList.txt el vocabulario acordado y posteriormente extrae de la clase WoAOntology.class (clase padre que extiende de la clase *Ontology*) todas las palabras del vocabulario existentes y se realiza una comparación para que ambas listas contengan el mismo vocabulario.
- **contentVocabularyTest()**: en este test, que es relativamente distinto al resto, se compara que las variables del vocabulario contienen el mismo nombre en su contenido que en su nombre propio.
- **conceptTest()**: extrae del fichero conceptList.txt la lista de conceptos de la ontología acordada y busca en la carpeta ontology.Concept las clases que estén heredando de la clase Concept. Debido a que *AgentAction* también hereda de Concept, es necesario ubicar estas clases en una carpeta aparte, para evitar errores de que se mezclen ambas clases cuando se realiza esta comparativa. Una vez extraída la lista, la compara con la lista acordada.
- **actionTest()**: extrae del fichero actionList.txt la lista de acciones de la ontología acordada. Posteriormente, busca en la carpeta ontología y sus hijas todas las clases que hereden de *AgentAction*. Por último, compara ambas listas para que tengan los mismos componentes.
- **predicateTest()**: extrae del fichero predicateList.txt la lista de predicados de la ontología acordada. Posteriormente, busca en la carpeta ontología y sus hijas todas las clases que hereden de *Predicate*. Por último, compara ambas listas para que tengan los mismos componentes.

Para llevar a cabo estos casos de prueba, se generaron 3 clases auxiliares en la carpeta utilTools que contienen los siguientes métodos para dar soporte a los test principales de la ontología:

- **ClassFinder.java**: Contiene los métodos getSchemaInPackage. Estos métodos generan una lista a partir de recorrer la estructura del código, desde una ubicación de partida y extraer todas las clases que extiendan de una estructura concreta, por ejemplo, *Predicate*.
- **FileToList.java**: Esta clase contiene el método getWordsFromFile, que como su nombre indica, genera una lista de *String* a partir de un fichero concreto. Para llevarlo a cabo, se le inserta la ruta del fichero y mientras que se lee el fichero, se van guardando las palabras y eliminando los conectores, tanto espacios en blanco como interlineados como comas.
- **SameName.java**: mientras que el resto de las clases se utilizan para la extraer los ficheros de los acuerdos y comprobar que se ha respetado en la ontología, este soporta al quinto test de la ontología que valida el contenido del vocabulario. Debido a que el formato de escritura del nombre del vocabulario y su contenido tienen estándares distintos, se requiere de una conversión para comparar su similitud. El método matchVariableAndContent, a partir de una lista de variables, explora su existencia en la clase WoAOntology.class (ya que se realizaría en ejecución) y saca su contenido una a una, transformando el string del nombre de la variable de snake_case a CamelCase (a través del segundo método de la clase) y vuelve tanto el nombre propio como su atributo a minúsculas, para comparar que sean iguales.

Estas cuatro clases son indispensables para que los test ontológicos fuesen posibles. Para que estos test funcionasen correctamente, se añadió en el pom.xml tres nuevas librerías en la sección de dependencias: org.reflections, org.codehaus.plexus y common-lang. La librería org.reflections se utiliza para escanear los metadatos y explorarlos en tiempo de ejecución, org.codehaus.plexus sirve para la manipulación de archivos y common-lang para la manipulación de las clases, todo ello necesario para poder extraer ficheros, ver contenido de clases y sus propiedades.

Gracias al método getWordsFromFile, los archivos de la carpeta *resource* pueden tener un formato variable, ya que permite tanto saltos de línea como espacios o limitadores como comas, por lo que ayuda a seguir manteniendo la flexibilidad que se busca en la práctica de la asignatura y deberá ser elegida por los alumnos el formato que van a seguir.

### 3.3.3 Test de integración

Para la regresión de pruebas, no podemos asegurar el correcto funcionamiento del programa. Además, existen ciertas funcionalidades, que no pueden probarse aisladas del resto. Los test de integración cumplen de la función de aumentar la cobertura de calidad creado pruebas sobre el conjunto, ayudando a encontrar problemas o errores que no son tan obvias durante las pruebas de los métodos o funciones [35]. En añadido, el conjunto del proyecto es un sistema multi-agente, lo que aumenta la complejidad de las pruebas, ya que es el propio agente el que a través de su comportamiento se comunicará con otros agentes y procesará la información para la toma de decisiones.

En estos test de integración lo que se busca es comprobar la correcta comunicación entre los agentes Tribu y el agente Plataforma en los primeros momentos del despliegue, comprobando que el entorno se genera y pueden comunicarse entre ellos sin errores asociados. Por ello, estos test no utilizan la librería de JUnit, sino que utilizamos un enfoque distinto.

Para ello, lo primero que se ha hecho es modificar la clase Main.java del esqueleto del proyecto, y añadir un nuevo tipo de despliegue. Normalmente, cuando se ejecuta un sistema multi-agente en Jade, se despliega el RMA (Resource Managment Agent) con el comando -gui, que genera la interfaz para gestionar y trabajar con los agentes [36]. Esto provoca un fallo en el despliegue de los pipelines que genera en errores a la hora de la ejecución con el runner. Por ello, se añadió un despliegue con un comando que despliega sin interfaz, compatible con los pipelines.

Los dos nuevos comandos son:

- **-td** o **--testdebug**: Ejecuta los agentes JADE propios de la compilación sin interfaz.
- **-tb** o **--testbuild**: Ejecuta los agentes de los .jar generados sin interfaz.

A su vez, se les asignó un tiempo de ejecución con la indicación de que debían cerrarse correctamente una vez transcurrido dicho plazo, con el fin de analizar únicamente el despliegue y los primeros momentos de la comunicación entre los agentes.

Una vez desarrollados los comandos mencionados, se añadieron al pipeline y se ejecutaron como parte de las pruebas de integración, lo que garantiza una cobertura desde el inicio del despliegue para asegurar el correcto funcionamiento de los agentes sin errores. Además, al estar incluidos como nuevos comandos, permiten a los equipos ejecutarlos localmente para comprobar su funcionamiento antes de subirlos a la rama.

## 3.4 Adaptación proyecto en IntelliJ

Además del esqueleto de Maven, los profesores de la asignatura tenían un segundo esqueleto del proyecto creado con un fichero .iml. Este fichero, configura el editor de código IntelliJ para adaptarlo a las necesidades del proyecto. Los proyectos asociados con IntelliJ y su .iml tenía una estructura diferente al proyecto con Maven. Maven tenía un directorio raíz, del que derivaba src que a su vez salía dos carpetas: Main y test. Mientras que, en el segundo esqueleto, test esta fuera de la carpeta src, por lo que algunos test, como los de las ontologías, como el pipeline, adaptado a trabajar con Maven y su estructura, hacía que fuese un enfoque distinto y requiriese, sobre todo en la parte de la integración con el pipeline (al no tener una herramienta de compilación incorporada), un segundo proyecto no contemplado al comienzo del trabajo de fin de grado.

Aun así, se realizaron ajustes al .iml y se actualizaron las librerías y pruebas unitarias, para que el segundo esqueleto pudiese ejecutar de manera local la compilación y la ejecución de las pruebas unitarias y de integración.

Como se puede observar en el Anexo 5 la adaptación a todas las librerías necesarias para la ejecución de las pruebas, mientras que en el fichero OntologyTest.java se adaptaron todas las referencias a la búsqueda y extracción de vocabulario, clases y ficheros para trabajar con las nuevas referencias y el funcionamiento en IntelliJ. Por último, se obtuvieron las nuevas librerías y se añadieron a la carpeta lib, de la que recoge el iml las referencias.

## 3.5 Migración e integración al Gitlab de la asignatura

Una vez probadas todas las funcionalidades en el GitLab personal, realizó la migración al GitLab de la asignatura. Se realizó un fork del Proyecto “Jade World of Agents skeleton Maven version” y se creó una rama en la que importar el proyecto. Por otro lado, se creó y registró un runner para el proyecto y la ejecución del pipeline.

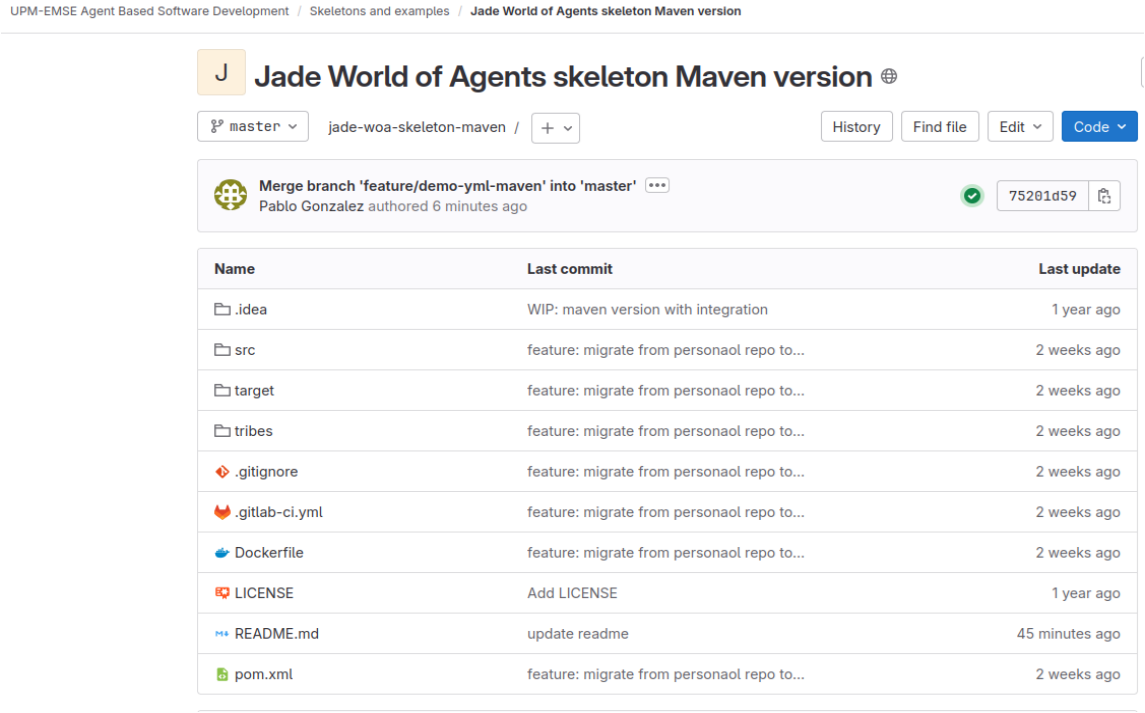
Una vez importado el proyecto, se procedió a subir la rama para la comprobación del pipeline y la ejecución de las pruebas tanto unitarias, ontológicas como de integración. Una vez subido se comprobó el éxito del despliegue.

A su vez, se migró en una rama del proyecto “Jade World of Agents skeleton Gradle version” la adaptación del .iml y todos los recursos nuevos añadidos, excluyendo el pipeline.

## 3.6 Demo

A continuación, se muestra los esqueletos de ambos proyectos cómo resultados visibles del desarrollo del trabajo.

Proyecto Maven




Readme del proyecto:

README.md

# EMSE-ABSD JADE World of Agents (Maven version)

License MIT

## Introduction

JADE Project for ABSD groups ready to be imported from Git as Gradle-build project in Intellij IDEA. The project includes both platform and tribe containers with a naive tribe agent trying to communicate with the platform agent. The project also includes the basic functionality to load agents from other tribes as .jar imports.

## Requirements

- Java 1.8 (JDK 17 recommended)
- JUnit 5.8.1
- Maven 3.8.8
- Lombok 1.18.30 (optional for reduce Java verbosity, refer to https://projectlombok.org)

## Structure

- `lib` : folder for libraries
- `out` : .jar artifacts to export the agents and run the game.
- `src.es.upm.emse.absd` : root module with the .java files to develop
  - `teamX` : module directory of each group.
  - `Main.java` : configurable main class that launches the project and imported agents from other teams.
  - `Utils.java` : Utility class to implement reusable functions.
- `test` : JUnit testbed.
- `tribes` : folder for importing tribe agents from other teams as libraries.

## Exaple of use

## Exaple of use

```
> java -jar woa-team0.jar -h
---------------------------------------
--- EMSE-ABSD JADE World of Agents ---
---------------------------------------
Agents:
    + Platform:
        - AgPlatform: An agent that responds when its called by its name.
    + Tribe:
        - AgTribe: An agent that try to talk with his/her colleague.
Usage: java -jar woa.jar [options]
    + options:
        -h, --help  Prints help
        -d, -debug  Run JADE agents from the compilation
        -b, -build  Run agents from generated jars
        -td, -testdebug  Run JADE agents from the compilation without interface and close the executi
        -tb, -testbuild  Run agents from generated jars without interface and close the execution 10
```

## Install GitLab runner (For ubuntu)

1. Add the gitlab repository:
   `curl -L "https://packages.gitlab.com/install/repositories/runner/gitlab-runner/script.deb.sh" | sudo bash`
2. Install GitLab runner
   `sudo apt-get install gitlab-runner`
3. Check version:
   `sudo gitlab-runner -version`

> ```
> Version:      16.11.0
> Git revision: 91a27b2a
> Git branch:   16-11-stable
> GO version:   go1.21.9
> Built:        2024-04-18T19:21:08+0000
> OS/Arch:      linux/amd64
> ```

4. Checkea el status
   `sudo gitlab-runner status`

## Register runner

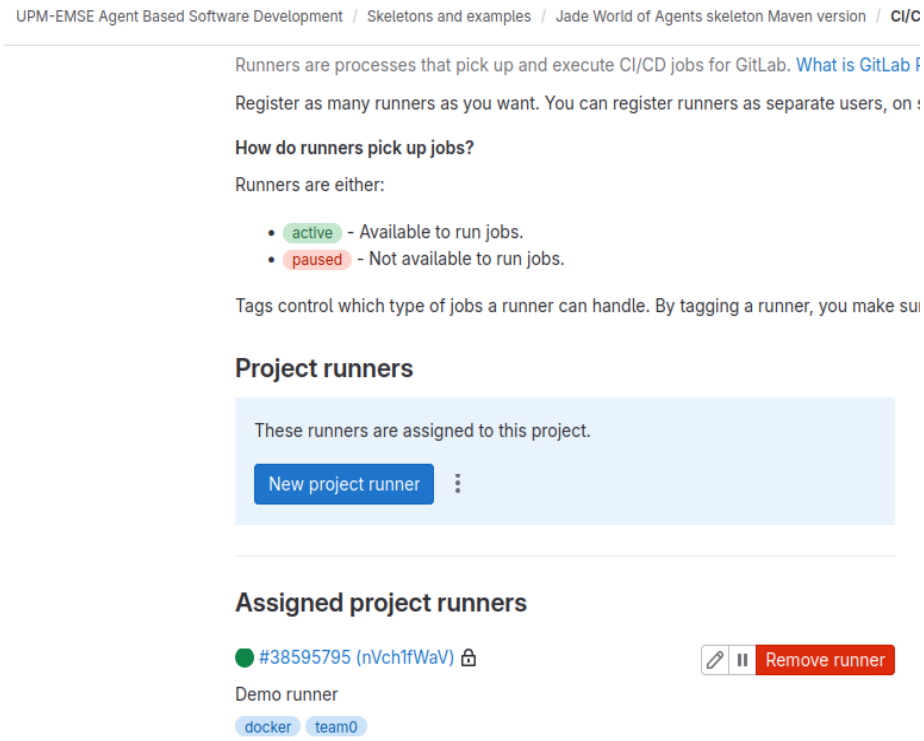
1. In the sidebar of the gitlab project, click **Settings>>CI/CD**
2. A screen with different sections will appear. Click on the **expand** button in the **Runners** section.
   - On the left side, project runners, click the button ***New project runner***.
3. A screen with different sections will appear that contain the following fields - Tags: - Tags: can be blank - Run untagged jobs: enable - Configuration (optional) - Runner description: write the name of the runner (*keep the name for later*) - Paused: disable - Protected: disable - Lock to current projects: enable - Maximun job timeout: blank
   - Click **Create runner**
4. In the next screen: - Select Platform
5. Open the terminal - Copy the command of the Gitlab screen and add **sudo** before: - `sudo gitlab-runner register --url ....` - Enter the Gitlab instance URL: press enter - Enter a name for the runner: Enter the same of the runner description (*step 3, 3rd check*) - Enter an executor: docker - Enter default Docker image: openjdk:17
6. Check the runner is created: `sudo gitlab-runner run`

The runner should appear in green in the project runners section.

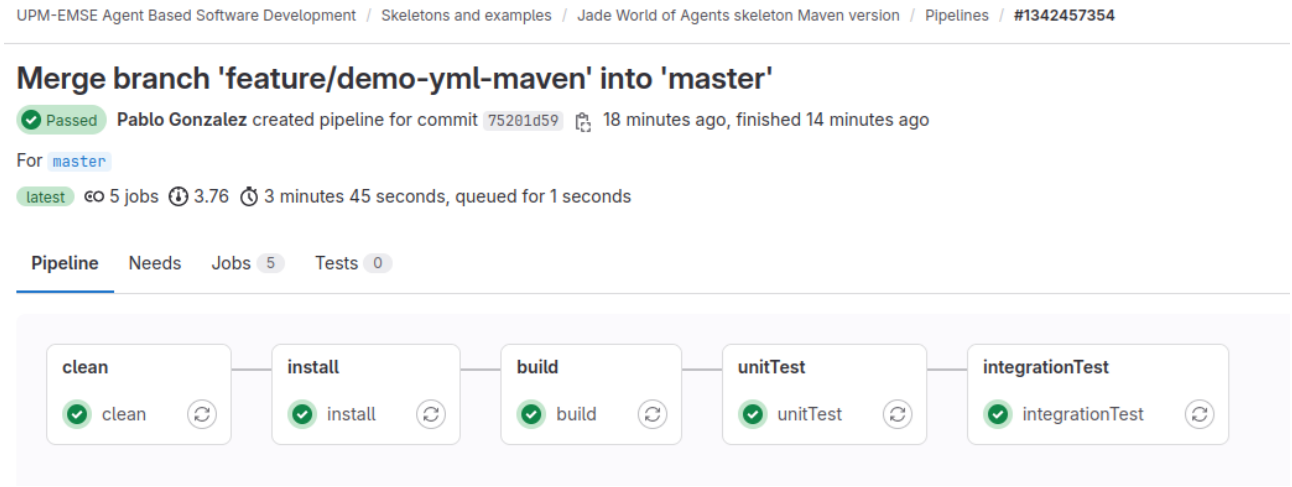
Note: If the runner is with a black triangle with the message: runner has never contacted this instance run `sudo gitlab-runner verify`

## Copyright and licence

Runner del proyecto registrado:

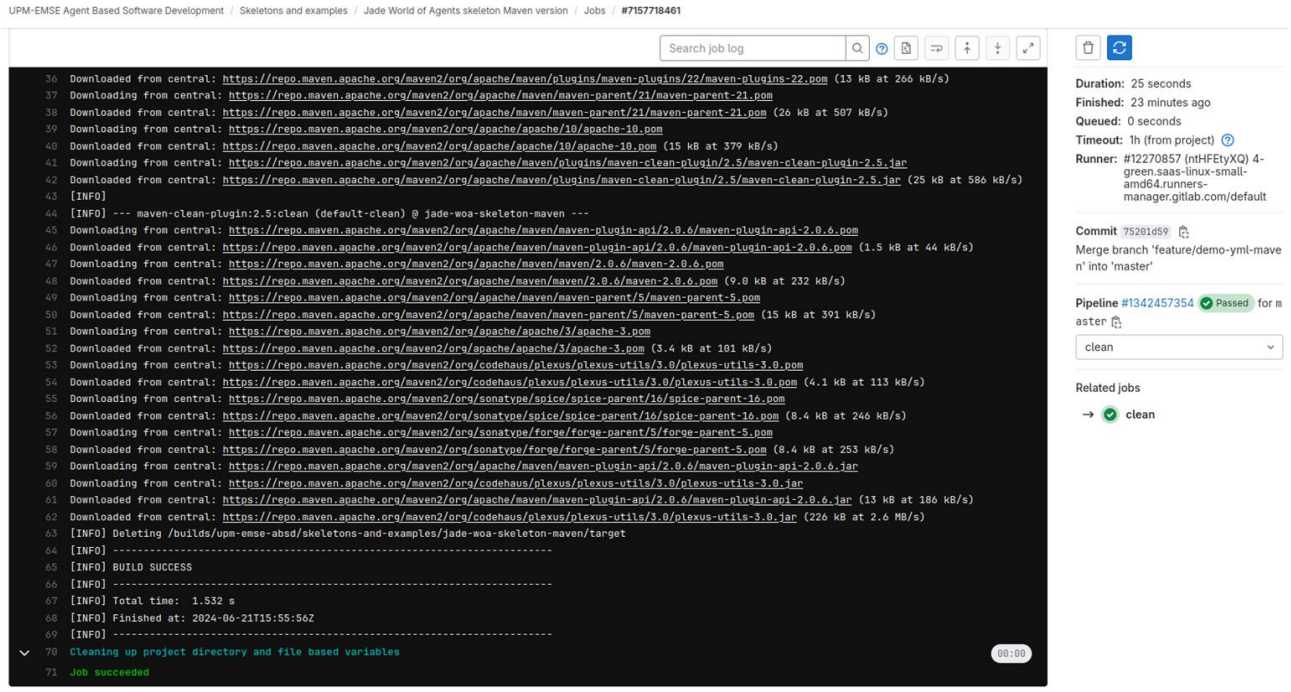



Pipeline del *merge* a la rama master:

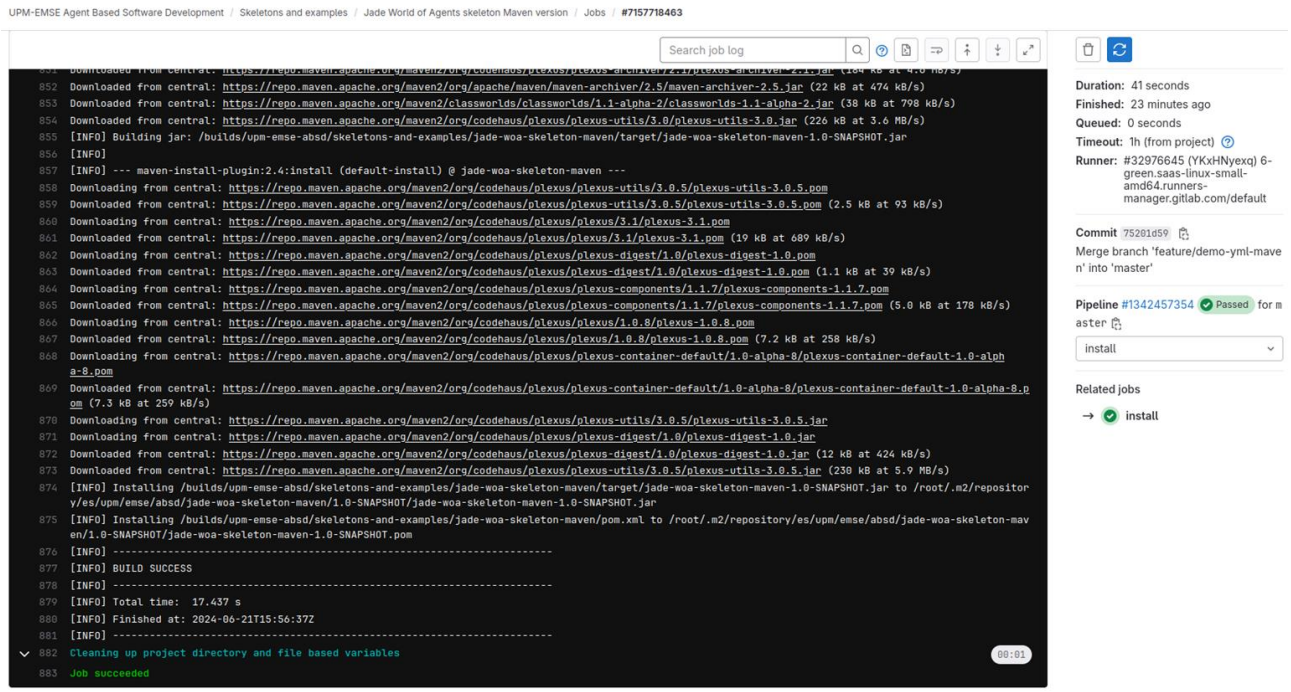


Se muestra una sección de cada step del pipeline:

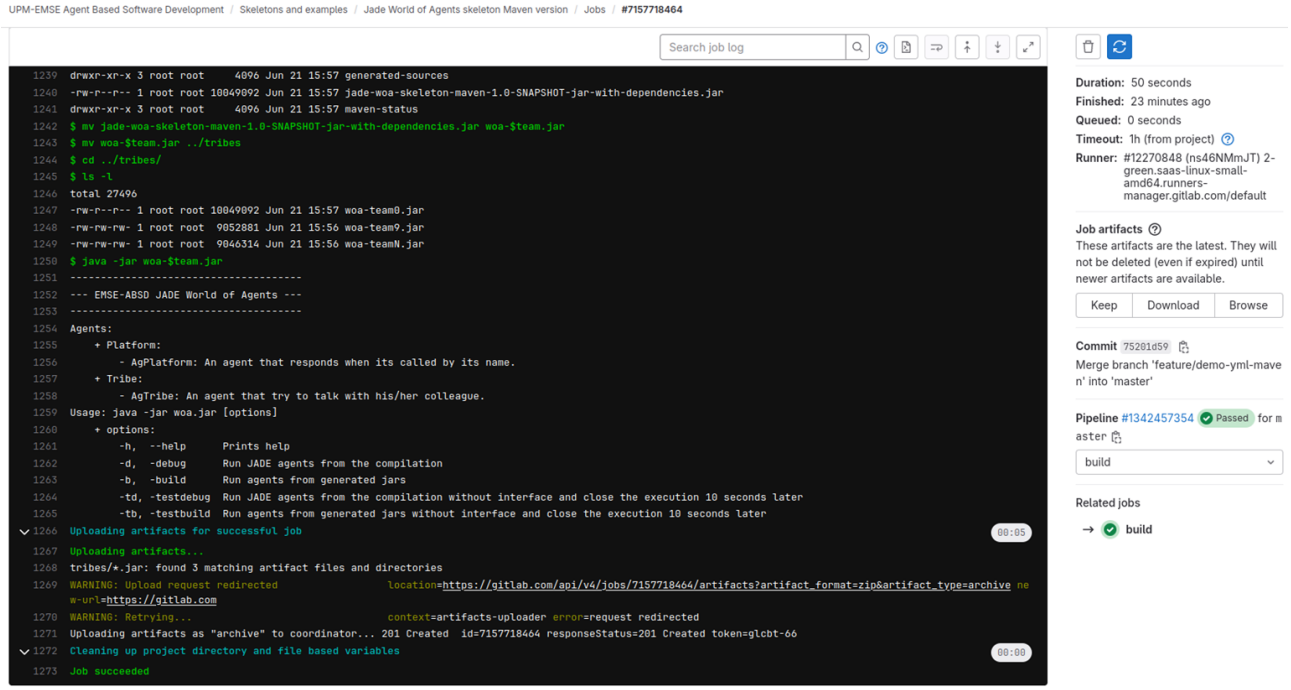
*Step* Clean



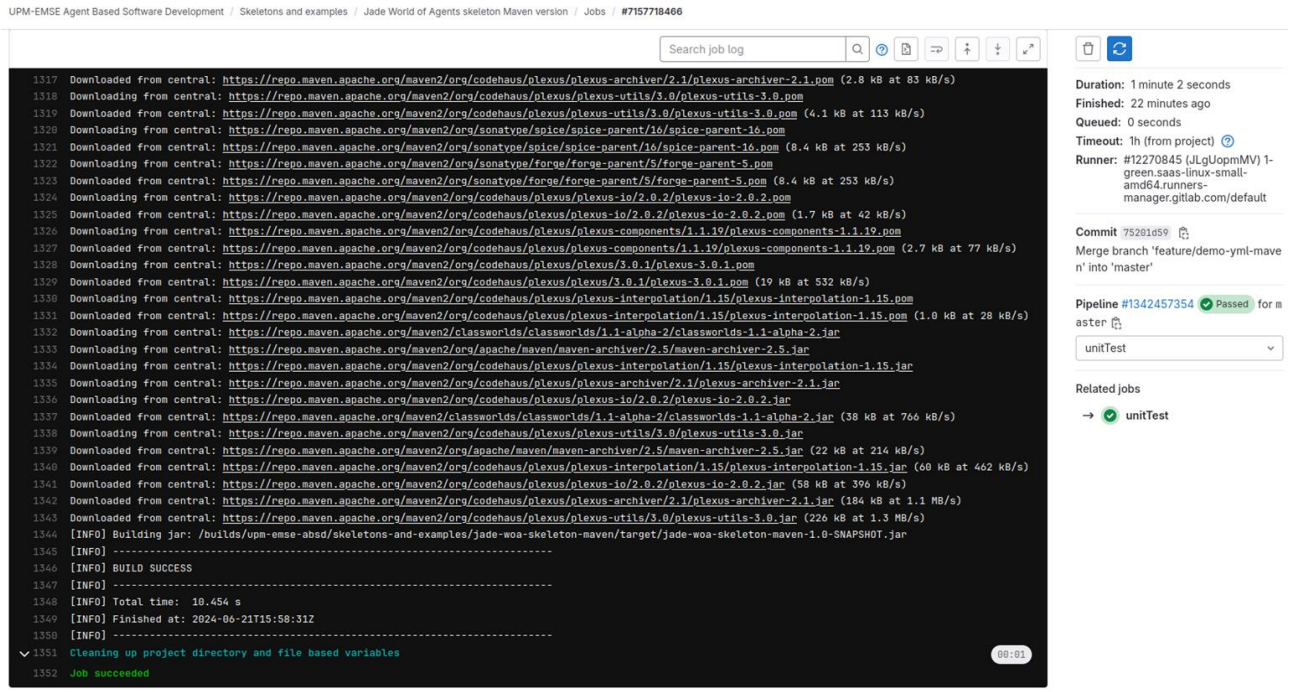

*Step* Install



*Step* Build: se puede observar la compilación exitosa con la ejecución del help

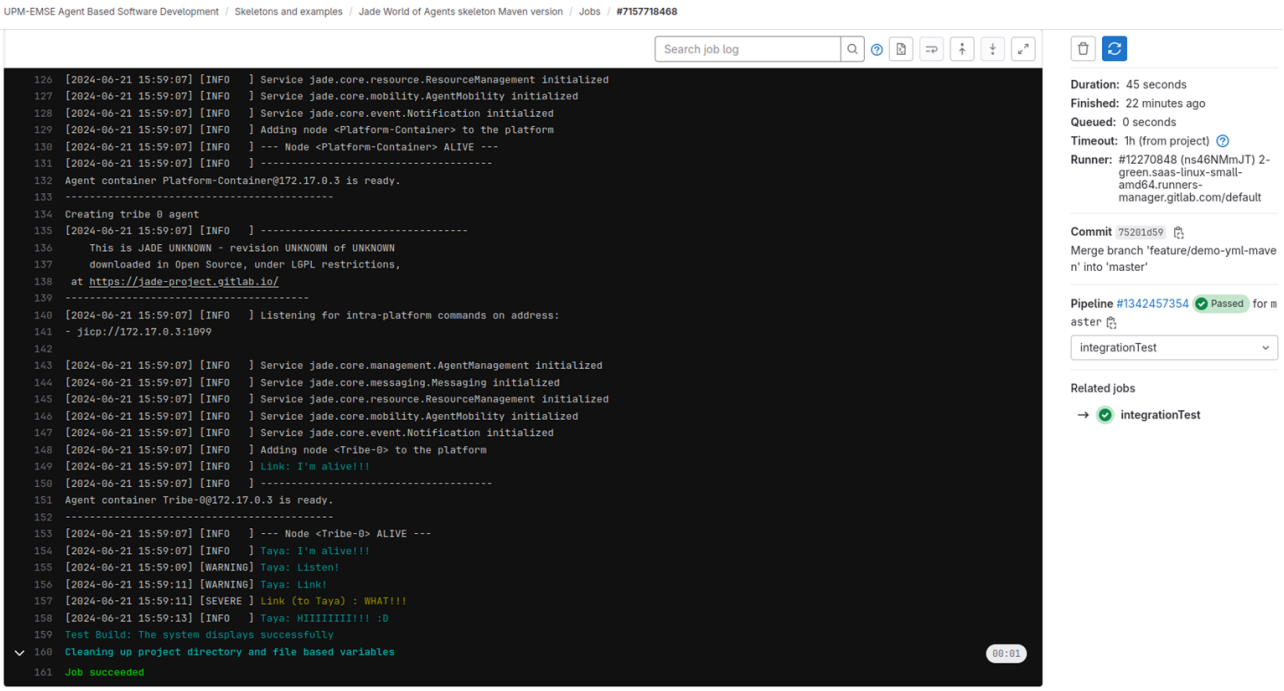



*Step* unitTest

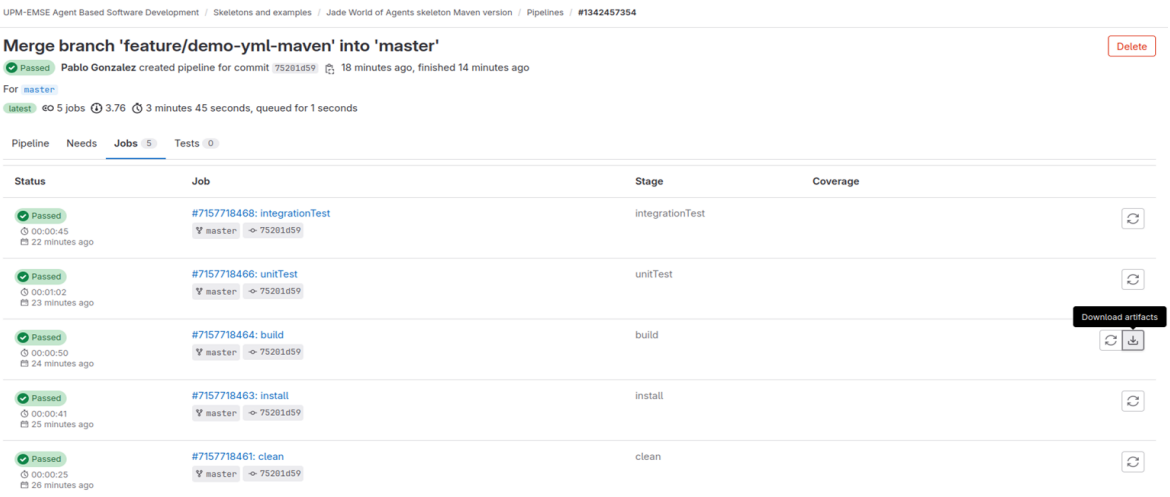


*Step* unitTest: se ve el resultado correcto de la ejecución de los test tanto unitarios como ontológicos

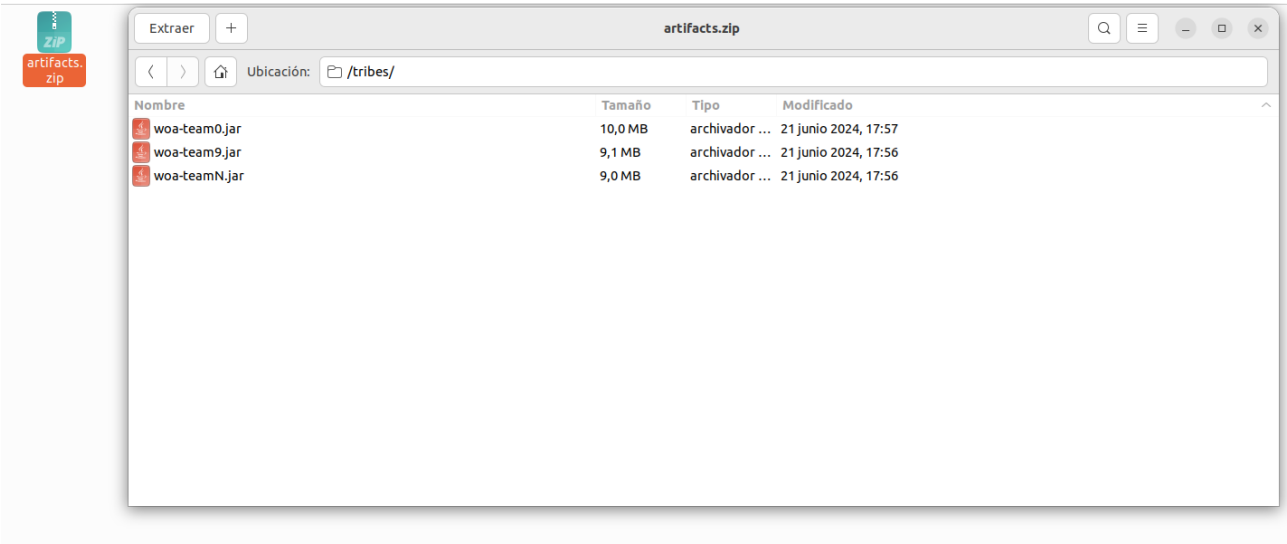

*Step* integrationTest




Pipeline ventana de Jobs: mostrando el *artifact* de la carpeta tribe, en la que se encuentra además del .jar recientemente compilado, el de todos los demás equipos que se hayan almacenado en el proyecto.



Archivo zip descargado del pipeline



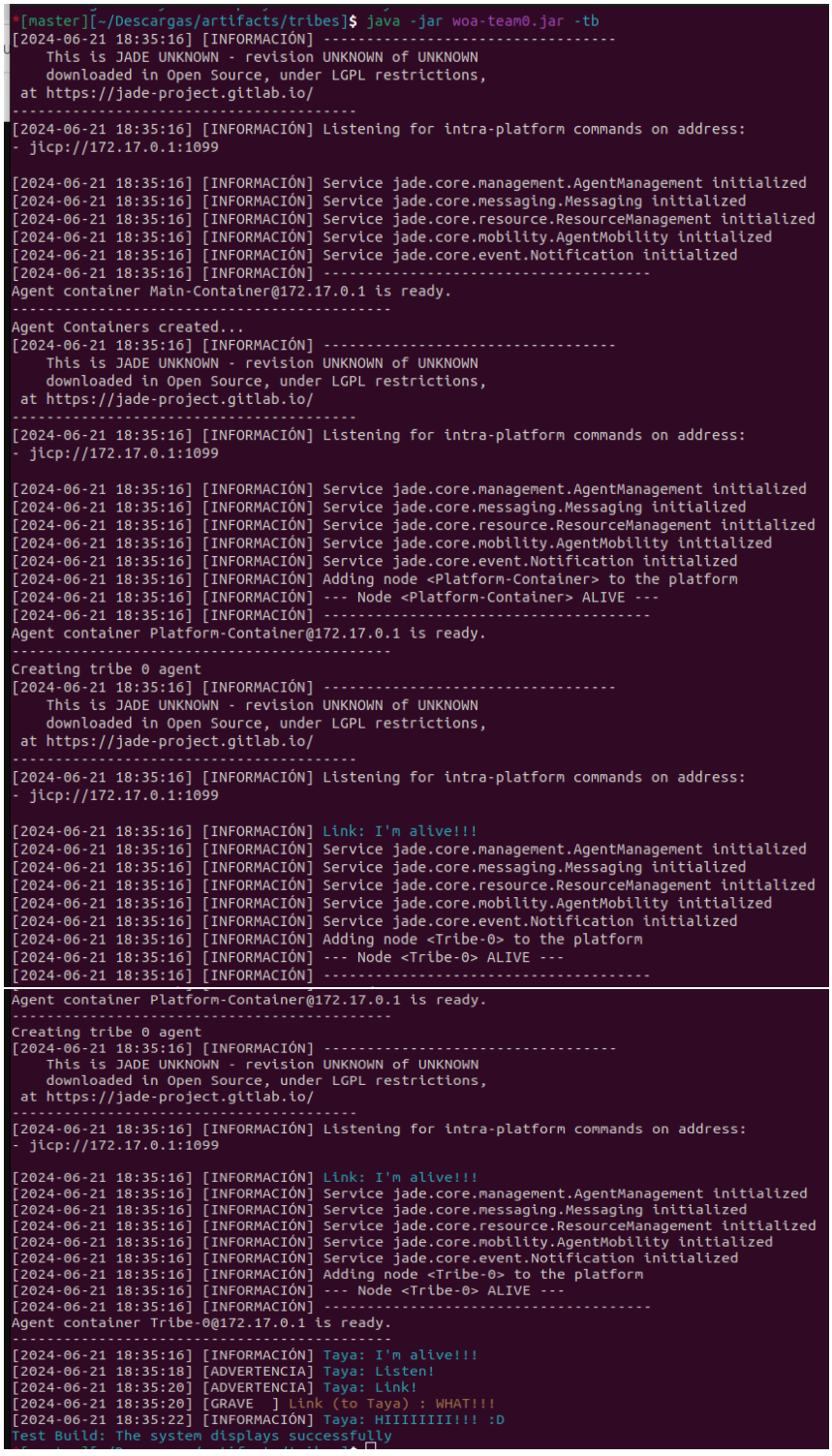

Ejecución del archivo .jar extraído del pipeline:
--help

```
*[master][~/Descargas/artifacts/tribes]$ java -jar woa-team0.jar
---------------------------------------
--- EMSE-ABSD JADE World of Agents ---
---------------------------------------
Agents:
    + Platform:
        - AgPlatform: An agent that responds when its called by its name.
    + Tribe:
        - AgTribe: An agent that try to talk with his/her colleague.
Usage: java -jar woa.jar [options]
    + options:
        -h,  --help      Prints help
        -d,  -debug      Run JADE agents from the compilation
        -b,  -build      Run agents from generated jars
        -td, -testdebug  Run JADE agents from the compilation without interface and close the execution 10 seconds later
        -tb, -testbuild  Run agents from generated jars without interface and close the execution 10 seconds later
```

Demostración del comando -td

```
*[master][~/Descargas/artifacts/tribes]$ java -jar woa-team0.jar -td
[2024-06-21 18:33:53] [INFORMACIÓN] ----------------------------------
    This is JADE UNKNOWN - revision UNKNOWN of UNKNOWN
    downloaded in Open Source, under LGPL restrictions,
 at https://jade-project.gitlab.io/
----------------------------------------
[2024-06-21 18:33:53] [INFORMACIÓN] Listening for intra-platform commands on address:
- jicp://172.17.0.1:1099

[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.management.AgentManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.messaging.Messaging initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.resource.ResourceManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.mobility.AgentMobility initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.event.Notification initialized
[2024-06-21 18:33:53] [INFORMACIÓN] ------------------------------------
Agent container Main-Container@172.17.0.1 is ready.
--------------------------------------------
Agent Containers created...
[2024-06-21 18:33:53] [INFORMACIÓN] ----------------------------------
    This is JADE UNKNOWN - revision UNKNOWN of UNKNOWN
    downloaded in Open Source, under LGPL restrictions,
 at https://jade-project.gitlab.io/
----------------------------------------
[2024-06-21 18:33:53] [INFORMACIÓN] Listening for intra-platform commands on address:
- jicp://172.17.0.1:1099

[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.management.AgentManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.messaging.Messaging initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.resource.ResourceManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.mobility.AgentMobility initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.event.Notification initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Adding node <Platform-Container> to the platform
[2024-06-21 18:33:53] [INFORMACIÓN] --- Node <Platform-Container> ALIVE ---
[2024-06-21 18:33:53] [INFORMACIÓN] ------------------------------------
Agent container Platform-Container@172.17.0.1 is ready.
--------------------------------------------
[2024-06-21 18:33:53] [INFORMACIÓN] ----------------------------------
    This is JADE UNKNOWN - revision UNKNOWN of UNKNOWN
    downloaded in Open Source, under LGPL restrictions,
 at https://jade-project.gitlab.io/
----------------------------------------
[2024-06-21 18:33:53] [INFORMACIÓN] Listening for intra-platform commands on address:
- jicp://172.17.0.1:1099

[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.management.AgentManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Link: I'm alive!!!
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.messaging.Messaging initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.resource.ResourceManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.mobility.AgentMobility initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.event.Notification initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Adding node <Tribe-0> to the platform
[2024-06-21 18:33:53] [INFORMACIÓN] --- Node <Tribe-0> ALIVE ---
[2024-06-21 18:33:53] [INFORMACIÓN] ------------------------------------
Agent container Tribe-0@172.17.0.1 is ready.
--------------------------------------------
[2024-06-21 18:33:53] [INFORMACIÓN] ----------------------------------
    This is JADE UNKNOWN - revision UNKNOWN of UNKNOWN
    downloaded in Open Source, under LGPL restrictions,
 at https://jade-project.gitlab.io/
----------------------------------------
[2024-06-21 18:33:53] [INFORMACIÓN] Listening for intra-platform commands on address:
- jicp://172.17.0.1:1099

[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.management.AgentManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Link: I'm alive!!!
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.messaging.Messaging initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.resource.ResourceManagement initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.mobility.AgentMobility initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Service jade.core.event.Notification initialized
[2024-06-21 18:33:53] [INFORMACIÓN] Adding node <Tribe-0> to the platform
[2024-06-21 18:33:53] [INFORMACIÓN] --- Node <Tribe-0> ALIVE ---
[2024-06-21 18:33:53] [INFORMACIÓN] ------------------------------------
Agent container Tribe-0@172.17.0.1 is ready.
--------------------------------------------
[2024-06-21 18:33:53] [INFORMACIÓN] Navi: I'm alive!!!
[2024-06-21 18:33:55] [ADVERTENCIA] Navi: Listen!
[2024-06-21 18:33:57] [ADVERTENCIA] Navi: Hello!
[2024-06-21 18:33:59] [ADVERTENCIA] Navi: Hello!
[2024-06-21 18:34:01] [ADVERTENCIA] Navi: Listen!
Test Debug: The system displays successfully
*[master][~/Descargas/artifacts/tribes]$
```

Demostración del comando -tb

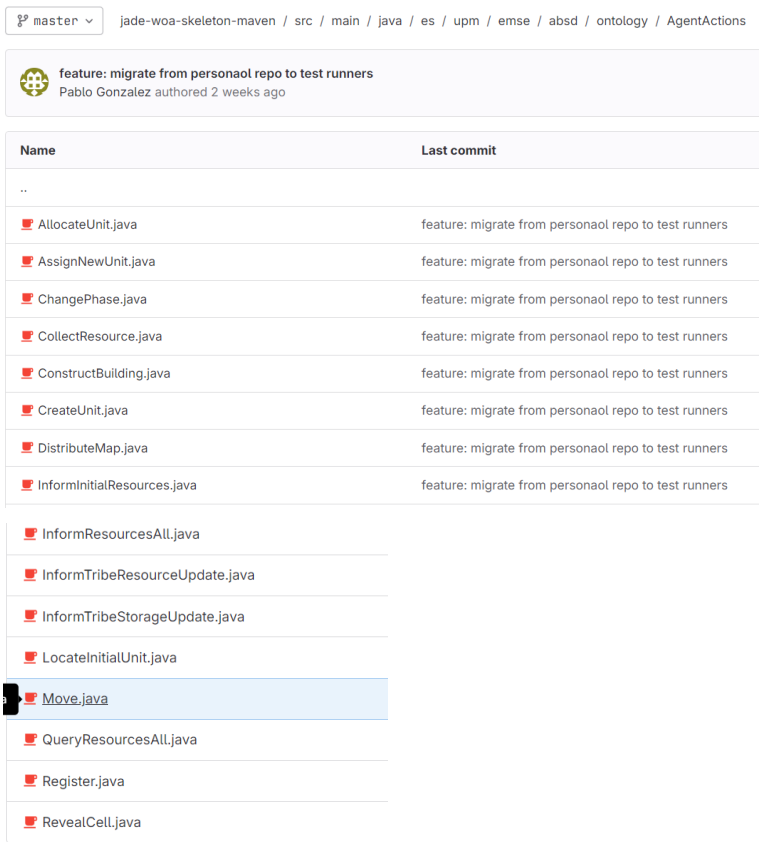


Ejemplo de lista de AgentActions acordadas correcta:

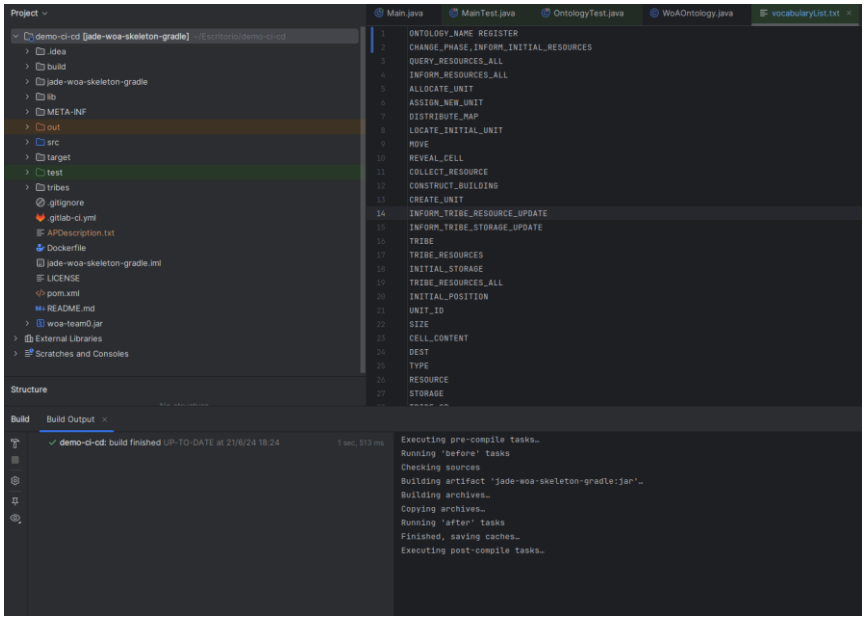
actionList.txt 254 B

```
1  CollectResource,
2  LocateInitialUnit
3  DistributeMap
4  InformTribeResourceUpdate
5  CreateUnit
6  ConstructBuilding
7  InformResourcesAll
8  RevealCell
9  Move
10 Register
11 InformTribeStorageUpdate
12 QueryResourcesAll
13 ChangePhase
14 AllocateUnit
15 AssignNewUnit
16 InformInitialResources
17
```

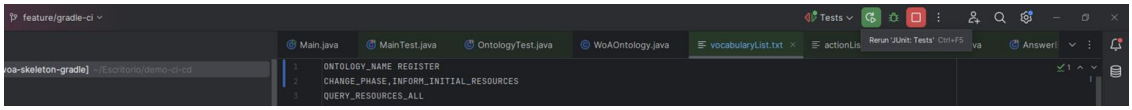
AgentActions añadidos a la ontología:



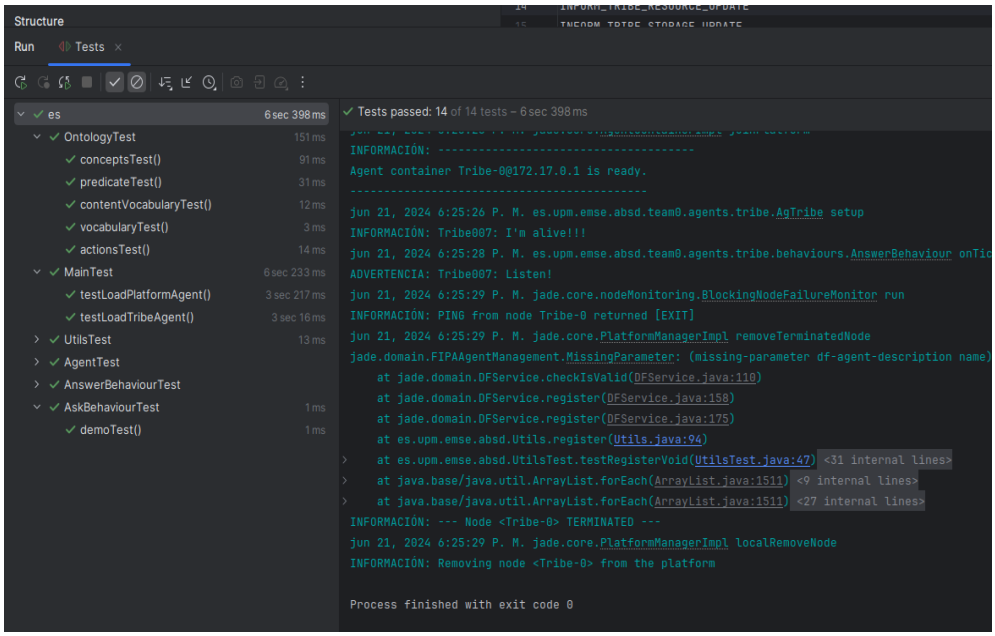
*Build* del 2o Proyecto en IntelliJ




Ejecución de los test




Resultados, se puede observar que el test ontológico citado en la página anterior pasa satisfactoriamente en el grupo de OntologyTest, la prueba actionTest:



## 3.7 Manual y documentación

Durante la creación del sistema de integración, se decidió el desarrollo de un manual, tanto para la instalación por parte de los equipos de los runners, cómo para la creación y mantenimiento de los test. En el Anexo 7.6 se encuentra subido el Manual completo. Ya que es una asignatura del máster universitario "European Master in Software Engineering", ha sido escrito en ingles para la facilitación del uso de todos sus alumnos.

Este manual tiene dos partes. Por un lado, contiene los pasos para la instalación y registro del runner en un proyecto GitLab.

Por otro lado, contiene documentación sobre las siguientes implementaciones:

- El fichero gitlab-ci.yml : explicando brevemente su contenido y funcionamiento.
- La estructura de test, cada una de las tres secciones descritas en el apartado 3.3 Creación de los test con un breve resumen de su contenido, estructura de carpetas y funcionamiento.

El objetivo de este manual es primero de todo facilitarles la instalación del runner para que puedan hacer uso de los pipelines y, en segundo lugar, que tengan un documento de apoyo de cara a la actualización de los test, según avancen la práctica de la asignatura. Por último, si algún alumno quisiese adaptar el pipeline que tuviese un contexto adecuado de su funcionamiento.

# 4 Resultados y conclusiones

## 4.1 Aspectos positivos

Tras un análisis de la infraestructura existente en la segunda práctica de Agent-Based Software Development del European Master in Software Engineering, se propusieron y desarrollaron medidas efectivas para mejorar la infraestructura, incluyendo la mejora en los recursos disponibles y en la implementación de nuevas tecnologías. A continuación, se presentarán los resultados obtenidos, destacando las principales aportaciones que ha supuesto este proyecto:

1. Se logró desplegar un runner funcional en GitLab que permitió la ejecución de pipelines de manera eficiente.

2. Se completó con éxito un pipeline en Maven para la automatización del proceso de compilación y pruebas del proyecto.

3. Se desarrollaron estructuras de test unitarios, de ontología y de integración para garantizar la calidad del código y su correcto funcionamiento.

4. Se elaboró un manual detallado para los alumnos de la asignatura, que incluye los pasos para la instalación del runner e información sobre el fichero gitlab-ci y las pruebas correspondientes.

5. Se llevó a cabo la actualización del proyecto en IntelliJ para mantenerlo al día con los últimos cambios disponibles en el proyecto.

6. Se trasladó la nueva infraestructura al Gitlab de la asignatura para facilitar su próxima utilización.

En este Trabajo de Fin de Grado se lograron los objetivos propuestos, demostrando la importancia de contar con un runner funcional, un pipeline completo en Maven, y una estructura completa de pruebas para garantizar la calidad del software. La elaboración de un manual detallado permitirá a los alumnos de la asignatura comprender y poner en práctica los conceptos aprendidos en clase de manera efectiva. Además, la actualización del proyecto en IntelliJ garantiza su compatibilidad y eficacia con las últimas versiones y da un primer paso para nuevas vías de despliegue del pipeline. En resumen, este trabajo ha demostrado la importancia de la automatización de procesos en el desarrollo de proyectos, así como la relevancia de la implementación de herramientas y estructuras de calidad para garantizar un desarrollo eficiente y fiable.

## 4.2 Problemas encontrados y soluciones

Durante el desarrollo del proyecto, se encontraron múltiples bloqueos e impedimentos que dificultaron el avance, pero que permitieron un mayor conocimiento de la materia aprendida. Los principales problemas y sus soluciones fuero las siguientes:

1. A la hora de ejecutar los test de integración, un problema que surgió en la ejecución del pipeline fue que, una vez ejecutado, permanecía sin terminar de parar en ningún momento, por lo que el pipeline acababa fallando. Para este problema se plantearon dos posibles soluciones: la primera era la ejecución del comando timeout de BASH junto a la línea de comando de ejecución del .jar (`timeout 30s java -jar woa-$team.jar -b`), la cual mata el proceso una vez pasado el tiempo que le otorgas [37]. La segunda opción, que fue la creación de comando auxiliar (`-tb y -td`) en el cual la ejecución era exactamente igual, menos que una vez desplegado se lanzaba un `Thread.sleep(10000)` y posteriormente se cortaba la ejecución. Se opto por esta segunda debido a que era preferible insertarlo en el código del Main más que en el pipeline, les otorgaba una nueva función para utilizarlo a los alumnos también en local sin necesidad que adaptar el comando en terminal.
2. A la hora de la creación de los test en JUnit, generaban problemas con la compilación de Maven, imposibilitando la opción de ejecutarlos. Para solucionarlo era necesario la incorporación en el pom.xml del plugin Surefire. Este plugin es necesario para la ejecución de pruebas unitarias, ayudando en la compilación de estas [38].
3. A la hora de escribir el README del proyecto, se descubrió la página stackedit.io que facilitó enormemente su redacción, pudiendo avanzar a mayor velocidad, sin necesidad de tener que ir viendo el resultado una vez consolidado el código en la rama.

Estos fueron los problemas más significativos encontrados, a parte de pequeños matices y cambios en la ejecución de los comandos que para no extender no se mencionan en esta sección, aunque se hace una mención especial a la página web en la cual se hallaron gran parte de las soluciones, Stack Overflow, en la cual hay numerosas secciones para todo tipo de incidencias, tanto para ejecuciones de comando de Maven, JUnit como de infraestructura y soluciones a las instalaciones del runner o las tecnologías utilizadas en este proyecto.

## 4.3 Líneas futuras

Analizando los resultados, se ha visto, siguiendo el ciclo de mejora continua, futuras líneas de desarrollo y mejora para alcanzar resultados más eficientes y mejorar el rendimiento:

1. Instalación de uno o varios runners en un servidor de la escuela, evitando la instalación por parte de los equipos y mejorando la eficiencia del desarrollo.
2. Integración del despliegue del proyecto en el pipeline o en un segundo pipeline para la presentación final de la práctica de la asignatura.
3. Ampliación de las pruebas de integración y unitarias.
4. Implementación de un pipeline adaptado para Gradle, otorgándoles otras vías compatibles para el desarrollo.
5. Implementación de un pipeline adaptado a la estructura de IntelliJ y el fichero .iml.

Con estas líneas de desarrollo, se pretende llegar en un futuro a la independencia completa o casi completa de la construcción y montaje del proyecto y a una mejora en la calidad del producto final por parte de los estudiantes.

# 5 Análisis de Impacto

Se espera que gracias a este desarrollo realizado obtengamos los siguientes beneficios desde el contexto de la asignatura, tanto de los alumnos como del profesorado:

1. Facilitación del proceso de trabajo y aumento en la eficiencia en el desarrollo de la práctica gracias a la implementación del runner.
2. Prevención de errores de compilación y ejecución, gracias a la automatización del proceso de compilación y de pruebas.
3. Disminución de errores de integración entre equipos de estudiantes de la asignatura gracias a las pruebas de integración.
4. Mejora de calidad y fiabilidad del código gracias a la implementación de un conjunto de pruebas y a su integración en el pipeline.
5. Mejora de la comprensión y manejo de la nueva infraestructura gracias al manual de instalación del runner y la documentación de los nuevos ficheros.
6. La migración al GitLab de la asignatura permite su inmediata utilización sin aumento de carga al profesorado y aumentando la accesibilidad y disponibilidad de este.
7. Mejora de la calidad de la enseñanza de la asignatura, ya que mejora la simulación de un entorno real de trabajo.

Se han analizado los resultados en la Agenda 2030 y podrían ser positivo en varios aspectos. En primer lugar, al automatizar el proceso de compilación y pruebas del proyecto con el uso de un pipeline en Maven, se está contribuyendo a la eficiencia y la sostenibilidad de los procesos de desarrollo de software. Esto puede ayudar a reducir el consumo de recursos y a minimizar la huella ambiental de las actividades relacionadas con el desarrollo de software, como se menciona en el objetivo 9.

Además, al desarrollar estructuras de test unitarios, ontológicos y de integración para garantizar la calidad del código, se está contribuyendo a la meta de promover el desarrollo sostenible y la innovación tecnológica a través de prácticas responsables en el sector de la tecnología de la información y la comunicación.

Finalmente, el desarrollo del sistema de integración, el desarrollo de un manual detallado para los alumnos de la asignatura y la actualización del proyecto en IntelliJ también pueden contribuir a la meta de promover la educación de calidad y el aprendizaje a lo largo de la vida, como menciona el objetivo 4, al facilitar el acceso a recursos educativos actualizados y de calidad para los estudiantes. En resumen, estos resultados pueden ayudar a alcanzar los objetivos de la Agenda 2030 en tecnología y educación.

# 6 Bibliografía

[1] K. Kelley. “What is gitlab and how to use it? 2024 | simplilearn”. Simplilearn.com. Acceso en Junio de 2024. [En línea]. Disponible: https://www.simplilearn.com/tutorials/git-tutorial/what-is-gitlab#:~:text=GitLab%20is%20a%20web-based,management%20to%20monitoring%20and%20security

[2] I. S. Alquinta. “Maximiza tu productividad con GitLab”. Listopro Community. Acceso en Junio de 2024. [En línea]. Disponible: https://community.listopro.com/maximiza-tu-productividad-con-gitlab/#:~:text=GitLab:%20GitLab%20se%20destaca%20por,de%20proyectos,%20wikis%20y%20más

[3] “Get started with GitLab CI/CD | GitLab”. GitLab Documentation. Acceso en Junio de 2024. [En línea]. Disponible: https://docs.gitlab.com/ee/ci/

[4] “GitLab University”. GitLab University. [En línea]. Acceso en Junio de 2024. Disponible: https://university.gitlab.com/courses/gitlab-runners

[5] IT technology business solutions by AERIS. Acceso en Junio de 2024. [En línea]. Disponible: https://www.aeris-consulting.com/wp-content/uploads/2022/04/Docker.pdf

[6] “Docker overview | Docker Docs - Docker Documentation.” Acceso en Junio de 2024. [En línea]. Disponible: https://docs.docker.com/guides/docker-overview/

[7] “Docker y su arquitectura: Una inmersión profunda.” Acceso en Junio de 2024. [En línea]. Disponible: https://www.linkedin.com/pulse/docker-y-su-arquitectura-una-inmersi%C3%B3n-profunda-bol%C3%ADvar-calder%C3%B3n-oewye/

[8] L. Vogel and S. S. (c) 2013 .-. 2024 vogella GmbH, “Apache Maven Tutorial.” Acceso en Junio de 2024. [En línea]. Disponible: https://www.vogella.com/tutorials/ApacheMaven/article.html

[9] “Maven And Its Features | SevenMentor - SevenMentor | Best Training Provider In Pune.” Acceso en Junio de 2024. [En línea]. Disponible: https://www.sevenmentor.com/maven-and-its-features

[10] “What is Gradle? Why Do We Use Gradle? [Updated] - Simplilearn.com.” Acceso en Junio de 2024. [En línea]. Disponible: https://www.simplilearn.com/tutorials/gradle-tutorial/what-is-gradle#:~:text=Gradle%20is%20a%20build%20automation,help%20of%20build%20automation%20tools

[11] “JUnit - Overview.” Acceso en Junio de 2024. [En línea]. Disponible: https://www.tutorialspoint.com/junit/junit_overview.htm

[12] P. Kainulainen, “JUnit 5 Tutorial: Running Unit Tests With Maven - Petri Kainulainen.” Acceso en Junio de 2024. [En línea]. Disponible: https://www.petrikainulainen.net/programming/testing/junit-5-tutorial-running-unit-tests-with-maven/

[13] “GitLab CI/CD :: Magnolia CMS Docs.” Acceso en Junio de 2024. [En línea]. https://docs.magnolia-cms.com/paas/gitlab/

[14] “FAQ | Jade Site” Acceso en Junio de 2024. [En línea]. Disponible: https://jade.tilab.com/support/faq/

[15] “Foundation for Intelligent Physical Agents - Wikipedia, la enciclopedia libre.” Acceso en Junio de 2024. [En línea]. Disponible: https://es.wikipedia.org/wiki/Foundation_for_Intelligent_Physical_Agents

[16] “Figura | representación de un sistema multiagente”. Acceso en Junio de 2024. [En línea]. Disponible: https://www.researchgate.net/figure/Figura-1-Representacion-de-un-sistema-multiagente_fig1_262727691

[17] “Why did we build Visual Studio Code” Acceso en Junio de 2024. [En línea]. Disponible: https://code.visualstudio.com/docs/editor/whyvscode

[18] P. Falstad & B. de Backer “An Introduction to the Z Shell”. Acceso en Junio de 2024. [En línea]. Disponible: https://ftp.nice.ch/pub/next/unix/shell/_zsh.3.0.5.NIHS.bs/zsh.3.0.5.NIHS.bs/docs/_intro.a4.pdf

[19] “Oh My Zsh - a delightful & open source framework for Zsh - Oh My Zsh!”. Acceso en Junio de 2024. [En línea]. Disponible: https://ohmyz.sh/

[20] “Medium. Installing Linux (Ubuntu 20.04) on an external portable SSD and pitfalls to be aware of”. Acceso en Junio de 2024. [En línea]. Disponible: https://medium.com/geekculture/installing-linux-ubuntu-20-04-on-an-external-portable-ssd-and-pitfalls-to-be-aware-of-388294e701b5

[21] “IntelliJ IDEA - Wikipedia, la enciclopedia libre.” Acceso en Junio de 2024. [En línea]. Disponible: https://es.wikipedia.org/wiki/IntelliJ_IDEA

[22] “IntelliJ IDEA: el IDE líder para Java y Kotlin - JetBrains Acceso en Junio de 2024. [En línea]. Disponible: https://www.jetbrains.com/es-es/idea/

[23] POTDAR, Amit M., et al. “Performance evaluation of docker container and virtual machine”. Procedia Computer Science, 2020, vol. 171, p. 1419-1428. Acceso en Junio de 2024. [En línea]. Disponible: https://www.sciencedirect.com/science/article/pii/S1877050920311315 (URL)

[24] “Install Docker Engine on Ubuntu | Docker Docs - Docker Acceso en Junio de 2024. [En línea]. Disponible: https://docs.docker.com/engine/install/ubuntu/

[25] “GitLab Runner | GitLab.” Acceso en Junio de 2024. [En línea]. Disponible: https://docs.gitlab.com/runner/

[26] “Install GitLab Runner | GitLab.” Acceso en Junio de 2024. [En línea]. Disponible: https://docs.gitlab.com/runner/install/index.html

[27] “Gestión de proyectos en GitLab. – Oficina de Software Libre y Conocimiento Abierto - Oficina de Software Libre y Conocimiento Abierto.” Acceso en Junio de 2024. [En línea]. Disponible: http://www.uhu.es/osl/index.php/2020/03/24/gestion-de-proyectos-en-gitlab/

[28] G. Fedoseev, A. Degtyarev, O. Iakushkin, and V. Korkhov, “A continuous integration system for MPD Root: Deployment and setup in GitLab”. Acceso en Junio de 2024. [En línea]. Disponible: https://www.semanticscholar.org/paper/9265e6539624805bf3c506f62c52da8652d8230b

[29] “Executors | GitLab.” Acceso en Junio de 2024. [En línea]. Disponible: https://docs.gitlab.com/runner/executors/

[30] “docker compose | Docker Docs - Docker Documentation.” Acceso en Junio de 2024. [En línea]. Disponible: https://docs.docker.com/reference/cli/docker/compose/

[31] “Containerize an application | Docker Docs - Docker Documentation.” Acceso en Junio de 2024. [En línea]. Disponible: https://docs.docker.com/guides/workshop/02_our_app/

[32] “¿Qué son las pruebas unitarias?: explicación de las pruebas unitarias en AWS - Amazon Web Services, Inc”. Acceso en Junio de 2024. [En línea]. Disponible: https://aws.amazon.com/es/what-is/unit-testing/#:~:text=Una%20prueba%20unitaria%20es%20un,la%20l%C3%B3gica%20te%C3%B3rica%20del%20desarrollador

[33] D. Hamilton, “Mejores prácticas de pruebas unitarias automatizadas: cómo aprovechar al máximo su prueba - Parasoft.” Acceso en Junio de 2024. [En línea]. Disponible: https://es.parasoft.com/blog/unit-testing-best-practices/

[34] “Building Multi-Agent System with JADE | Using ontologies” Acceso en Junio de 2024. [En línea]. Disponible: https://www.iro.umontreal.ca/~vaucher/Agents/Jade/Ontologies.htm

[35] “Unit testing vs integration testing | CircleCI.” Acceso en Junio de 2024. [En línea]. Disponible: https://circleci.com/blog/unit-testing-vs-integration-testing/

[36] “Introduction to the RMA | Jade Site” Acceso en Junio de 2024. [En línea]. Disponible: https://jade.tilab.com/documentation/tutorials-guides/jade-rma/introduction-to-the-rma/

[37] “Linux run a command with a time limit (timeout)”. Acceso en Junio de 2024. [En línea]. Disponible: https://www.cyberciti.biz/faq/linux-run-a-command-with-a-time-limit/

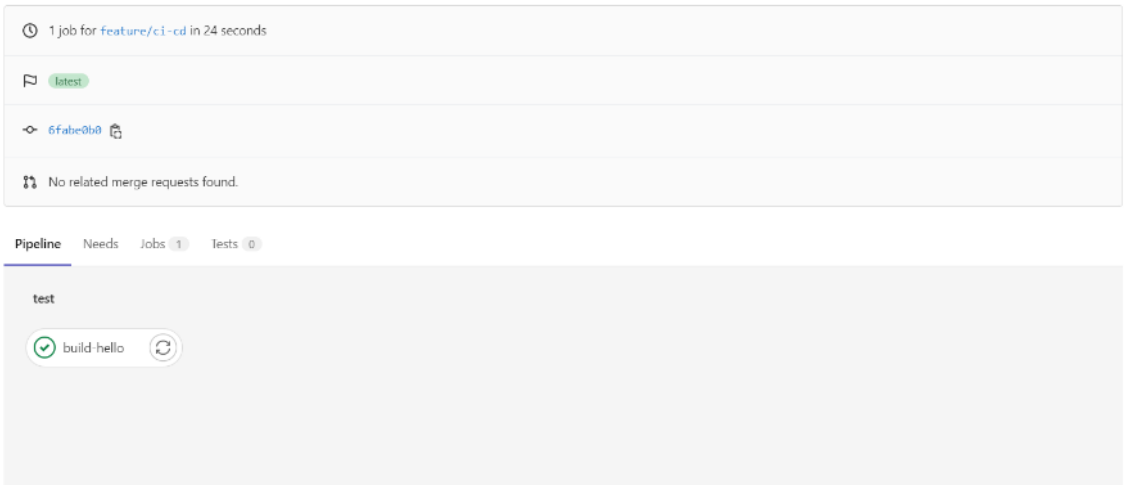
[38] S. C. Ramirez, “Maven Surefire Plugin – Introduction.”. Acceso en Junio de 2024. [En línea]. Disponible: https://maven.apache.org/surefire/maven-surefire-plugin/

# 7 Anexos

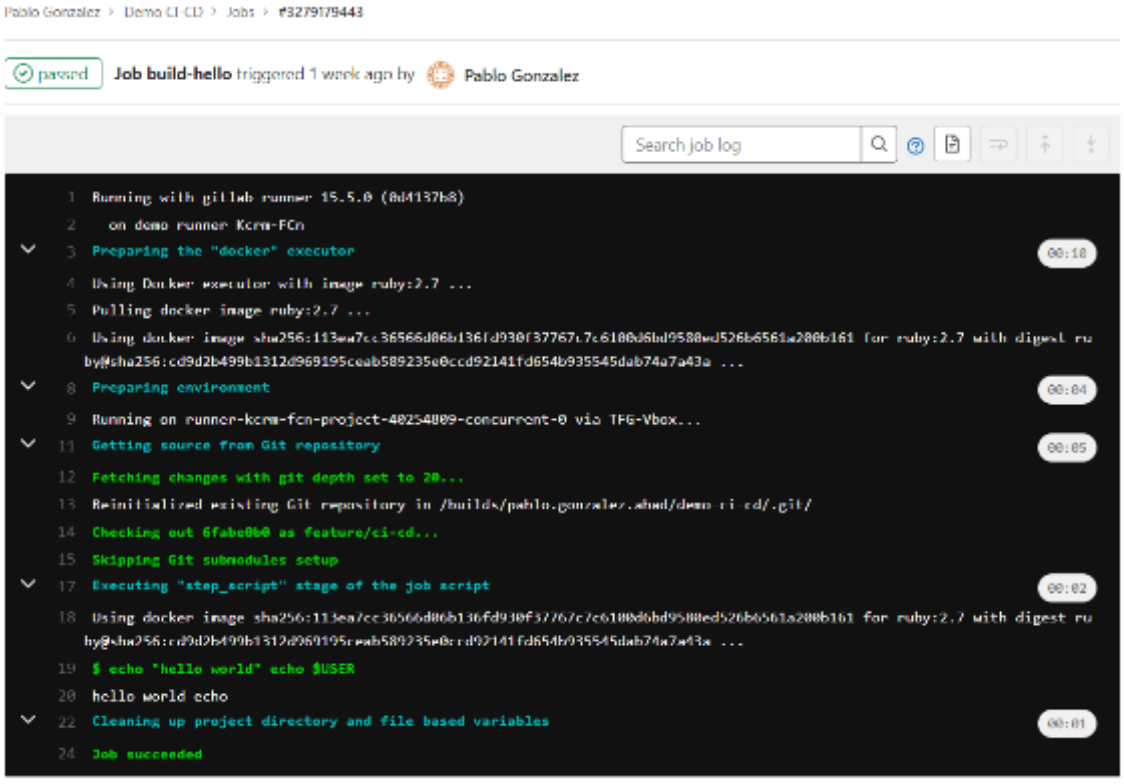
## 7.1 Anexo 1

Disponible en: https://gitlab.com/pablo.gonzalez.abad/demo-ci-cd/-/pipelines/689001739

Demo 1: Pipeline exitoso de la primera versión




Logs del runner en el pipeline de la primera versión

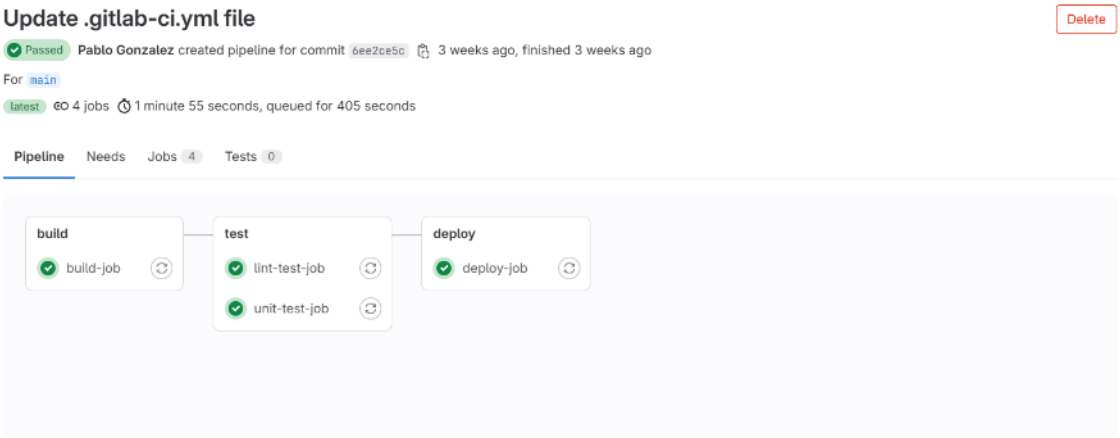


## 7.2 Anexo 2

Disponible en: https://gitlab.com/pablo.gonzalez.abad/demo-ci-cd/-/pipelines/1307409109

Demo 2: Segunda versión, explorando distintas opciones de gitlab-ci.yml

```
# This file is a template, and might need editing before it works on your project.
# This is a sample GitLab CI/CD configuration file that should run without any modifications.
# It demonstrates a basic 3 stage CI/CD pipeline. Instead of real tests or scripts,
# it uses echo commands to simulate the pipeline execution.
#
# A pipeline is composed of independent jobs that run scripts, grouped into stages.
# Stages run in sequential order, but jobs within stages run in parallel.
#
# For more information, see: https://docs.gitlab.com/ee/ci/yaml/index.html#stages
#
# You can copy and paste this template into a new `.gitlab-ci.yml` file.
# You should not add this template to an existing `.gitlab-ci.yml` file by using the `include:` keyword.
#
# To contribute improvements to CI/CD templates, please follow the Development guide at:
# https://docs.gitlab.com/ee/development/cicd/templates.html
# This specific template is located at:
# https://gitlab.com/gitlab-org/gitlab/-/blob/master/lib/gitlab/ci/templates/Getting-Started.gitlab-ci.yml

stages:          # List of stages for jobs, and their order of execution
  - clean
  - build
  - test
  - deploy
  - package

build-job:       # This job runs in the build stage, which runs first.
  stage: build
  script:
    - echo "Compiling the code..."
    - echo "Compile complete."

unit-test-job:   # This job runs in the test stage.
  stage: test    # It only starts when the job in the build stage completes successfully.
  script:
    - echo "Running unit tests... This will take about 60 seconds."
    - sleep 60
    - echo "Code coverage is 90%"

lint-test-job:   # This job also runs in the test stage.
  stage: test    # It can run at the same time as unit-test-job (in parallel).
  script:
    - echo "Linting code... This will take about 10 seconds."
    - sleep 10
    - echo "No lint issues found."

deploy-job:      # This job runs in the deploy stage.
  stage: deploy  # It only runs when *both* jobs in the test stage complete successfully.
  environment: production
  script:
    - echo "Deploying application..."
    - echo "Application successfully deployed."
```


Update .gitlab-ci.yml file
Delete
Passed
Pablo Gonzalez created pipeline for commit 6ee2ce5c 3 weeks ago, finished 3 weeks ago
For main
latest
4 jobs
1 minute 55 seconds, queued for 405 seconds
Pipeline
Needs
Jobs 4
Tests 0
build
build-job
test
lint-test-job
unit-test-job
deploy
deploy-job

## 7.3 Anexo 3

Disponible en: https://gitlab.com/pablo.gonzalez.abad/demo-ci-cd/-/commit/5a6d98138603b0927e1afea5334207af9a8a5ff0

Demo 3: tercera versión del yml, integración de Maven y docker
Gitlab-ci.yml

```
image: docker:latest
stages:          # List of stages for jobs, and their order of execution
  - clean
  - install
  - build
  - test
  - deploy
  - package


clean:
  stage: clean
  image: maven:3.8.8
  script:
    - mvn clean


install:
  stage: install
  image: maven:3.8.8
  script:
    - mvn install -Dmaven.test.skip=true
    - mvn verify
```

```
build:
  stage: build
  image: maven:3.8.8
  script:
    - mvn clean compile assembly:single
    - cd target
    - java -jar jade-woa-skeleton-maven-1.0-SNAPSHOT-jar-with-dependencies.jar
  artifacts:
    paths:
      - target/*.jar


deploy:
  stage: deploy
  image: docker
  services:
    - docker:dind
  before_script:
    - docker info
    - docker-compose --version
  script:
    - docker-compose -f docker-compose.yml down
    - docker-compose -f docker-compose.yml up -d
```

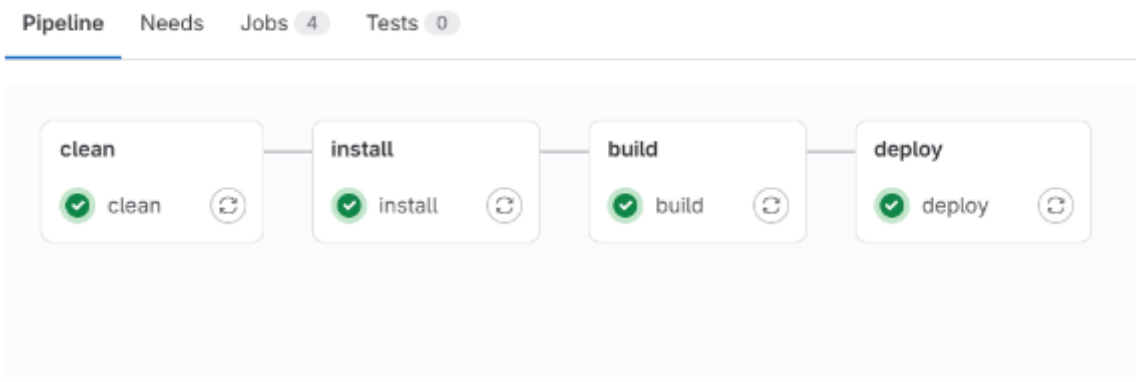
Dockerfile

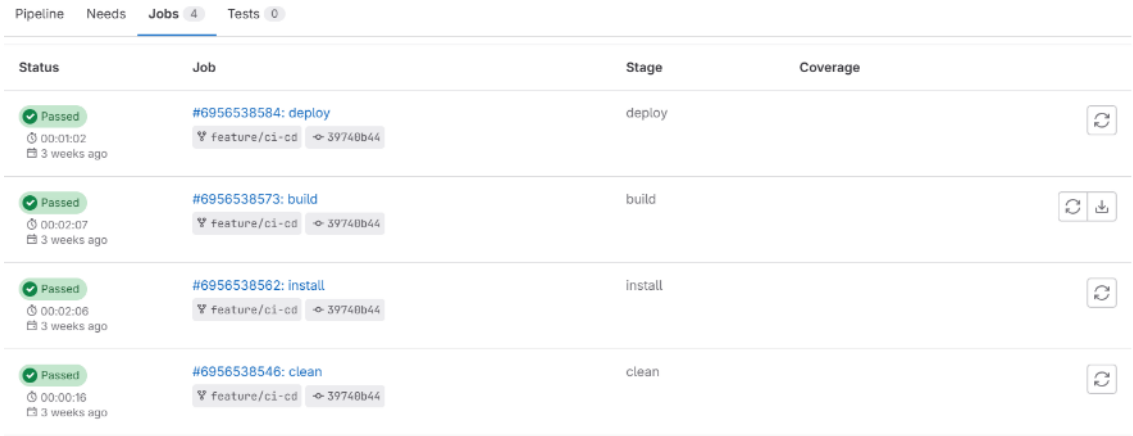
```
# syntax=docker/dockerfile:1

FROM openjdk:17
FROM maven:3.8.8
WORKDIR /app
COPY . /app
RUN mvn -v
RUN mvn clean
RUN mvn compile assembly:single -e
CMD ["java", "-jar", "/tribes/jade-woa-skeleton-maven-1.0-SNAPSHOT.jar"]
```

Docker-compose.yml

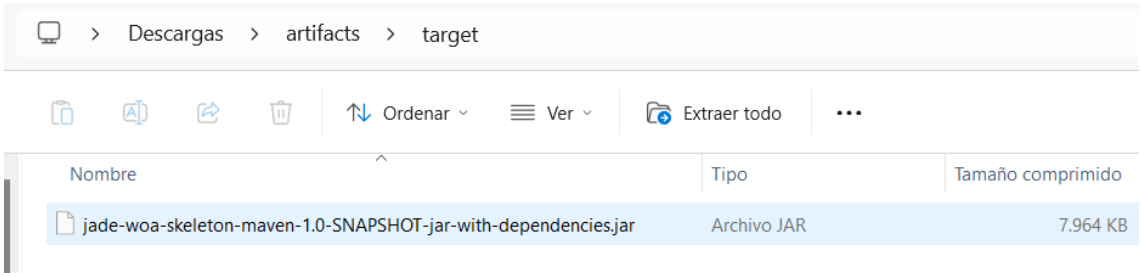
```
services:
  web:
    build: .
    ports:
      - "8080:8080"
  backend:
    image: "maven:3.8.8"
```

Ejecución del pipeline:

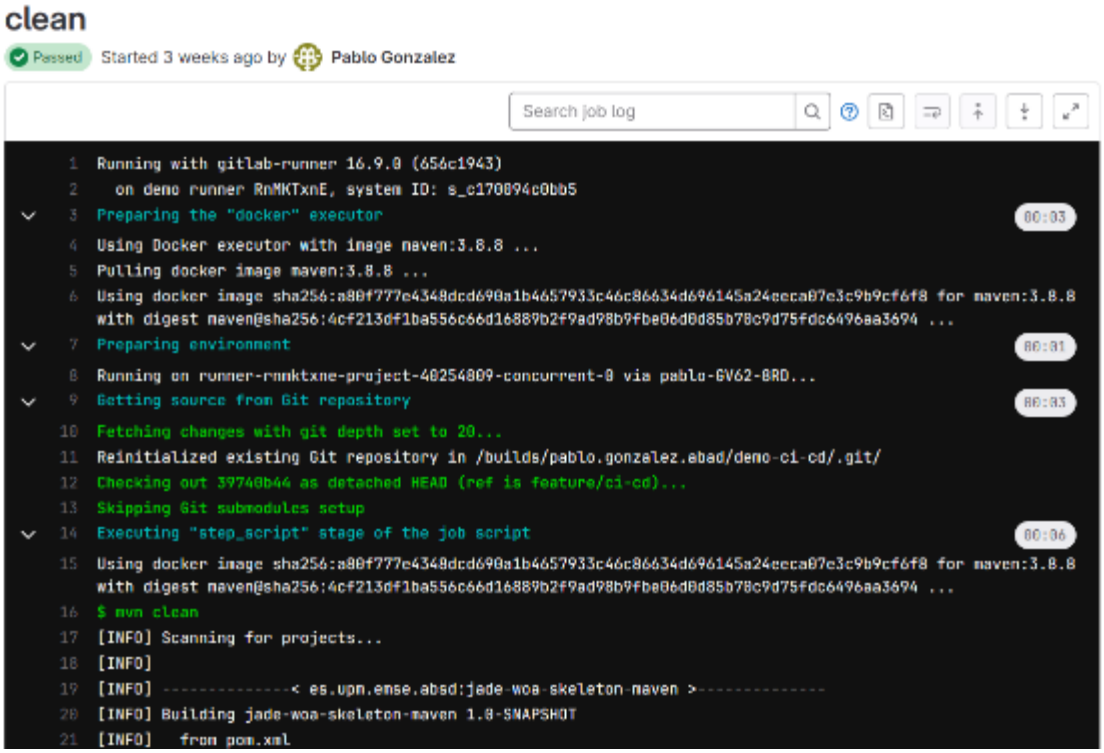




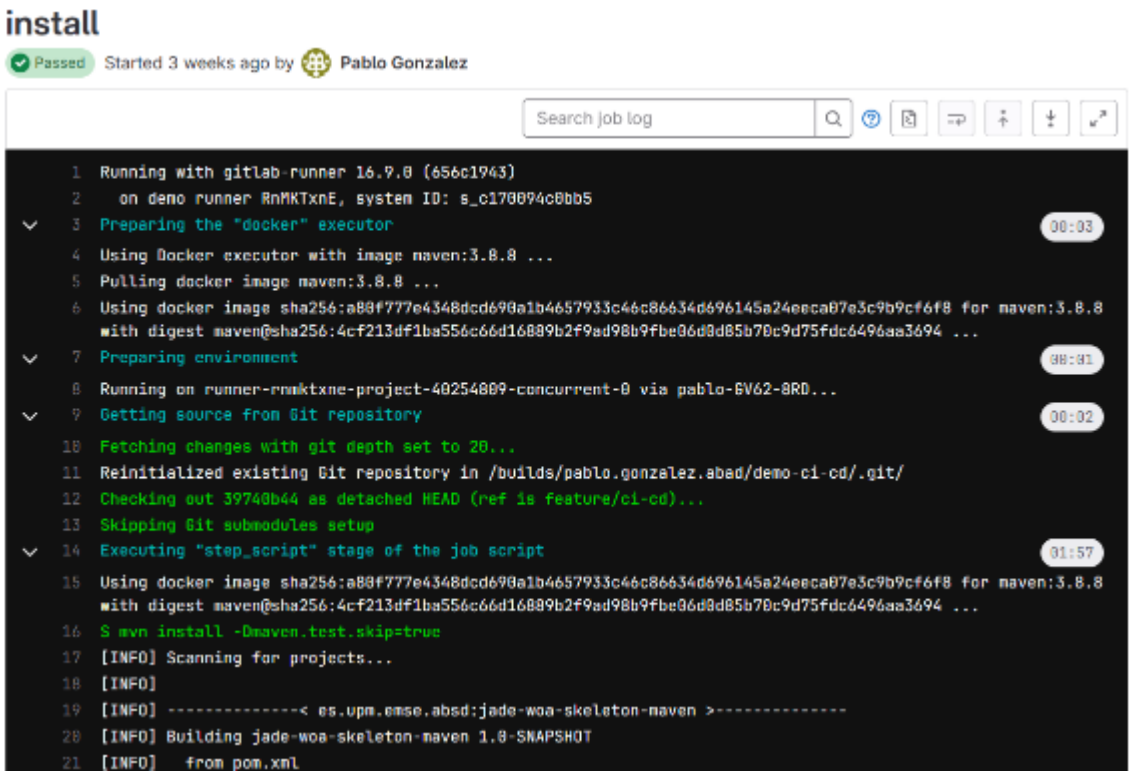

Archivo descargado del artifact de esa misma ejecución:



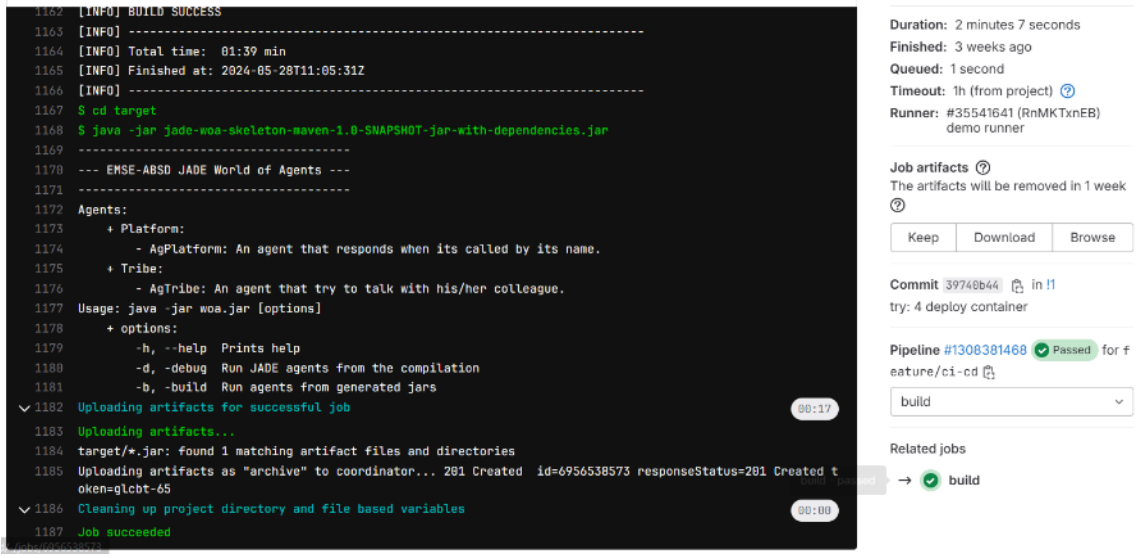

Step: Clean



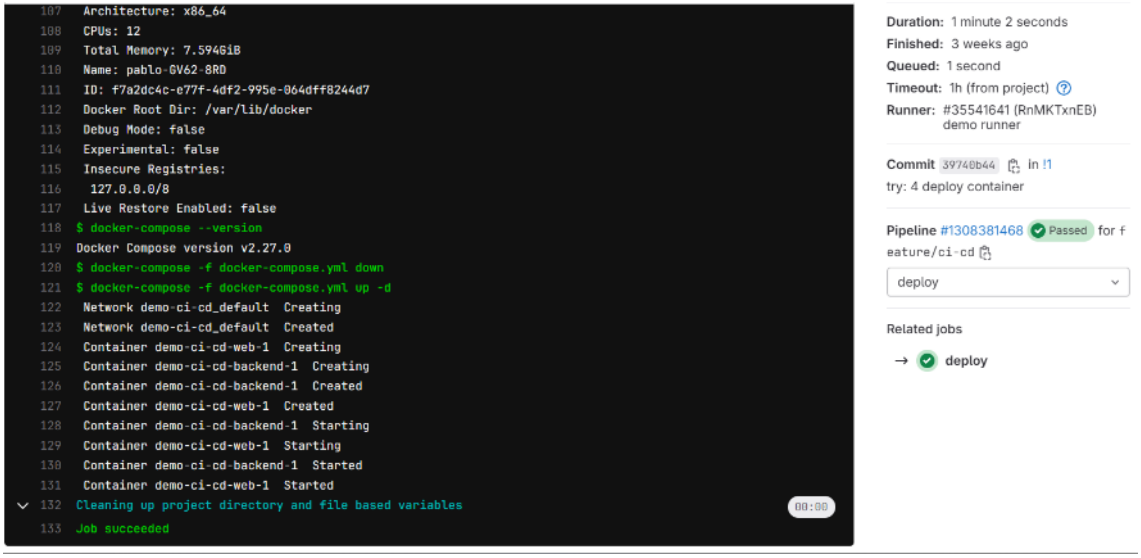
Step: Install




Step: Build



Step: Deploy




Sección actualizada del Pom.wml

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0
http://maven.apache.org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
…
    <dependencies>
        <dependency>
            <groupId>com.gitlab.jade-project</groupId>
            <artifactId>jade</artifactId>
            <version>master-SNAPSHOT</version>
        </dependency>
        <dependency>
            <groupId>org.projectlombok</groupId>
            <artifactId>lombok</artifactId>
            <version>1.18.30</version>
            <scope>provided</scope>
        </dependency>
…

    </dependencies>

</project>
```

## 7.4 Anexo 4

Disponible en: https://gitlab.com/pablo.gonzalez.abad/demo-ci-cd/-/pipelines/1319988745

Gitlab-ci.yml

```
image: docker:latest
image: maven:3.8.8

stages:          # List of stages for jobs, and their order of execution
  - clean
  - install
  - build
  - unitTest
  - integrationTest


clean:
  stage: clean
  script:
    - mvn clean


install:
  stage: install
  script:
    - mvn install -Dmaven.test.skip=true


build:
  stage: build
  script:
    - mvn clean compile assembly:single
    - cd src/main/java/es/upm/emse/absd/team*
    - team=${PWD##*/}
    - cd ../../../../../../../../target
    - ls -l
    - mv jade-woa-skeleton-maven-1.0-SNAPSHOT-jar-with-dependencies.jar woa-$team.jar
    - mv woa-$team.jar ../tribes
    - cd ../tribes/
    - ls -l
    - java -jar woa-$team.jar
  artifacts:
    paths:
      - tribes/*.jar
```

```
unitTest:
  stage: unitTest
  script:
    - mvn test
    - mvn verify
  allow_failure: true


integrationTest:
  stage: integrationTest
  script:
    - ls
    - cd src/main/java/es/upm/emse/absd/team*
    - team=${PWD##*/}
    - cd ../../../../../../../../tribes
    - ls
    - export DISPLAY=:0.0
    - java -jar woa-$team.jar -td
    - java -jar woa-$team.jar -tb
```

## 7.5 Anexo 5

Disponible: https://gitlab.com/upm-emse-absd/skeletons-and-examples/jade-woa-skeleton-gradle/-/tree/feature/gradle-ci?ref_type=heads

Jade-woa-skeleton-gradle.iml

```
<?xml version="1.0" encoding="UTF-8"?>
<module type="JAVA_MODULE" version="4">
  <component name="NewModuleRootManager" inherit-compiler-output="true">
    <exclude-output />
    <content url="file://$MODULE_DIR$">
      <sourceFolder url="file://$MODULE_DIR$/src" isTestSource="false" />
      <sourceFolder url="file://$MODULE_DIR$/test" isTestSource="true" />
    </content>
    <orderEntry type="inheritedJdk" />
    <orderEntry type="sourceFolder" forTests="false" />
    <orderEntry type="library" name="woa-teamDemo" level="project" />
    <orderEntry type="library" name="woa-team0" level="project" />
    <orderEntry type="library" name="lib" level="project" />
    <orderEntry type="library" name="jade-4.6.0" level="project" />
    <orderEntry type="module-library" scope="TEST">
      <library name="JUnit5.8.1">
        <CLASSES>
          <root url="jar://$MAVEN_REPOSITORY$/org/junit/jupiter/junit-jupiter/5.8.1/junit-jupiter-5.8.1.jar!/" />
          <root url="jar://$MAVEN_REPOSITORY$/org/junit/jupiter/junit-jupiter-api/5.8.1/junit-jupiter-api-5.8.1.jar!/" />
          <root url="jar://$MAVEN_REPOSITORY$/org/opentest4j/opentest4j/1.2.0/opentest4j-1.2.0.jar!/" />
          <root url="jar://$MAVEN_REPOSITORY$/org/junit/platform/junit-platform-commons/1.8.1/junit-platform-commons-1.8.1.jar!/" />
          <root url="jar://$MAVEN_REPOSITORY$/org/apiguardian/apiguardian-api/1.1.2/apiguardian-api-1.1.2.jar!/" />
          <root url="jar://$MAVEN_REPOSITORY$/org/junit/jupiter/junit-jupiter-params/5.8.1/junit-jupiter-params-5.8.1.jar!/" />
          <root url="jar://$MAVEN_REPOSITORY$/org/junit/jupiter/junit-jupiter-engine/5.8.1/junit-jupiter-engine-5.8.1.jar!/" />
          <root url="jar://$MAVEN_REPOSITORY$/org/junit/platform/junit-platform-engine/1.8.1/junit-platform-engine-1.8.1.jar!/" />
        </CLASSES>
        <JAVADOC />
        <SOURCES />
      </library>
    </orderEntry>
    <orderEntry type="library" name="junit-jupiter-5.8.1" level="project" />
    <orderEntry type="library" name="apiguardian-api-1.1.2" level="project" />
    <orderEntry type="library" name="lombok" level="project" />
    <orderEntry type="library" name="lombok" level="project" />
    <orderEntry type="library" name="woa-teamN" level="project" />
    <orderEntry type="library" name="javassist" level="project" />
    <orderEntry type="library" name="pfl-basic-5.0.0" level="project" />
    <orderEntry type="library" name="pfl-basic-tools-5.0.0" level="project" />
    <orderEntry type="library" name="javassist-3.28.0-GA" level="project" />
    <orderEntry type="library" name="jsr305-3.0.2" level="project" />
    <orderEntry type="library" name="reflections-0.10.2" level="project" />
    <orderEntry type="library" name="slf4j-api-1.7.32" level="project" />
  </component>
</module>
```

## 7.6 Anexo 6, Manual de instalación y documentación de los test

### 7.6.1 Runner installation

#### 7.6.1.1 Install GitLab runner (For Ubuntu)

1. Add the GitLab repository:
   ```
   $ curl -L
   "https://packages.gitlab.com/install/repositories/runner/gitl
   ab-runner/script.deb.sh" | sudo bash
   ```

2. Install GitLab runner
   ```
   $ sudo apt-get install gitlab-runner
   ```

3. Check version:
   ```
   $ sudo gitlab-runner -version
   ```

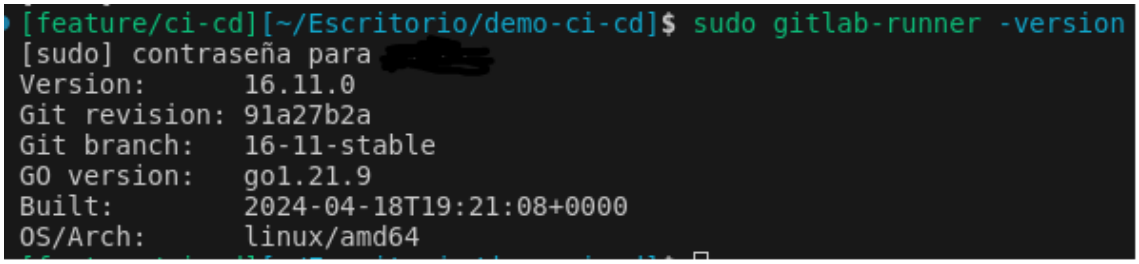



**Fig. 1** excetion of the –version command

4. Check the status
   ```
   $ sudo gitlab-runner status
   ```


**Fig. 2**execution of the status command

#### 7.6.1.2 Register runner

1. In the sidebar of the GitLab project, click **Settings>>CI/CD**

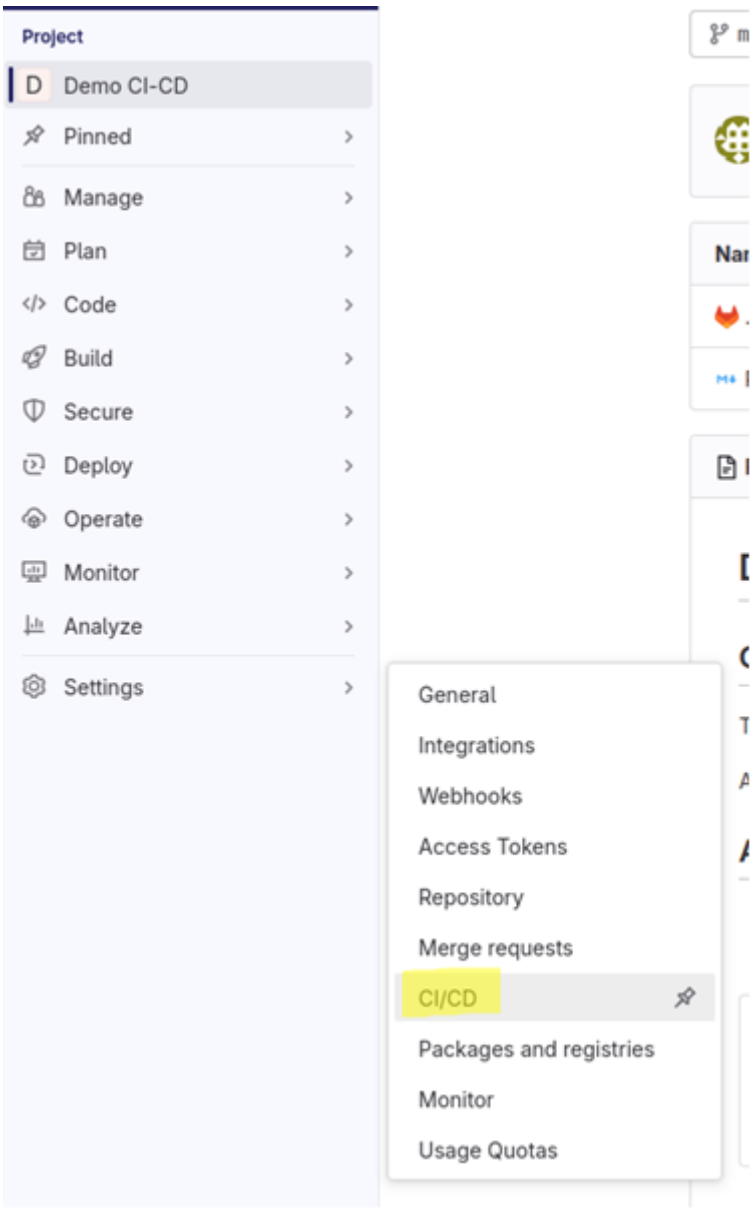



**Fig. 3** GitLab sidebar

2. A screen with different sections will appear. Click on the **expand** button in the **Runners** section.

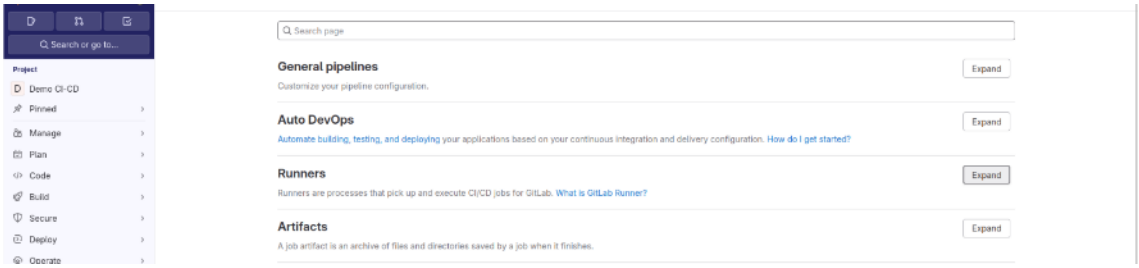



**Fig. 4** Setting CI/CD screen

On the left side, in the project runners section, click the button **New project runner**.

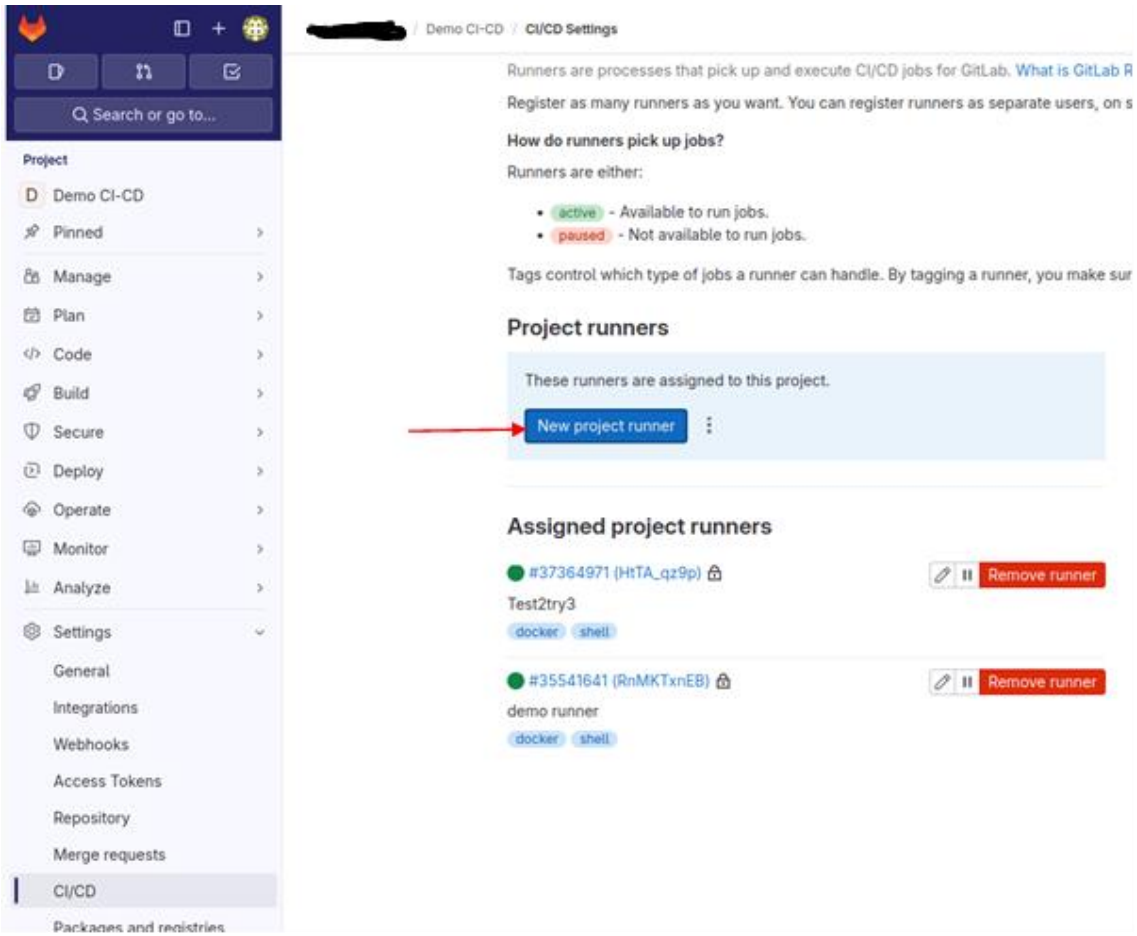



**Fig. 5** Runner section in Setting screen

3. A screen with different sections will appear that hold the following fields

Tags:

- **Tags**: To avoid problems with using runners from different teams, it is recommended to put the name of the team, for example team0. In this case

after each step in the gitlab-ci.yml file, place the following line of code after stage:
`tag: team0`

- **Run untagged jobs**: enable

Configuration (optional):

- **Runner description**: print the name of the runner (**keep the name for later**)
- **Paused**: disable
- **Protected**: disable
- **Lock to current projects**: enable
- **Maximum job timeout**: can be blank
- **Create runner**

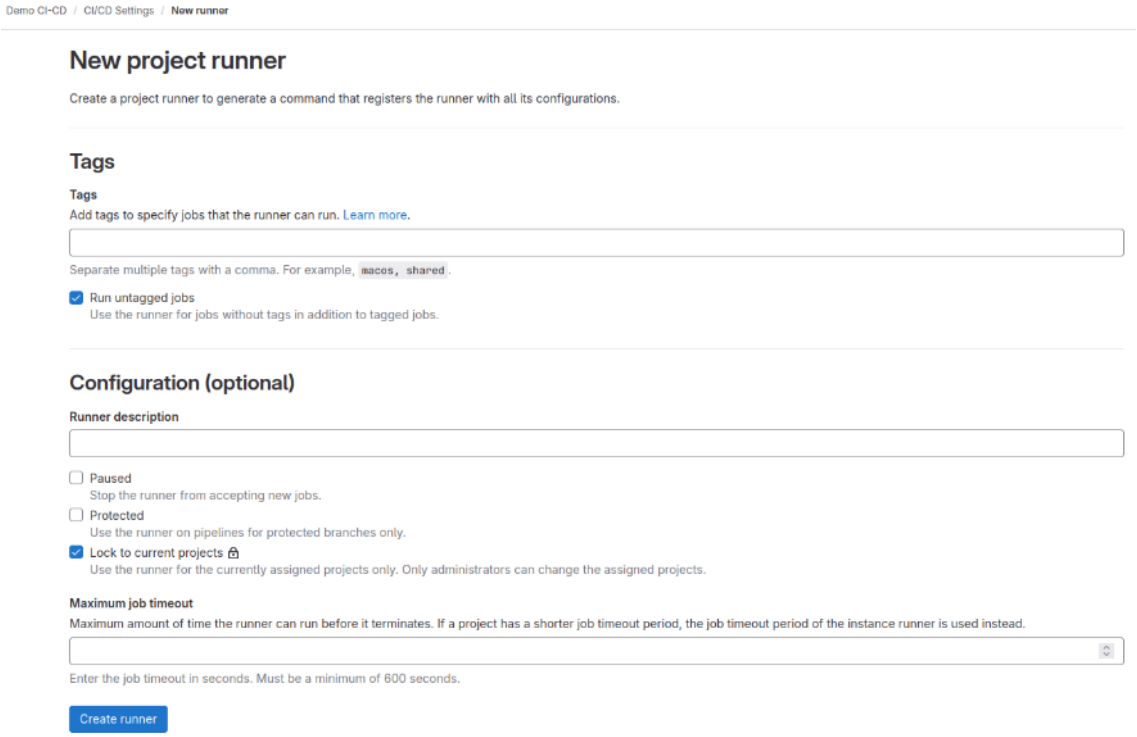



**Fig. 6** Screen 1 runner registration

4. Select Platform and open the terminal

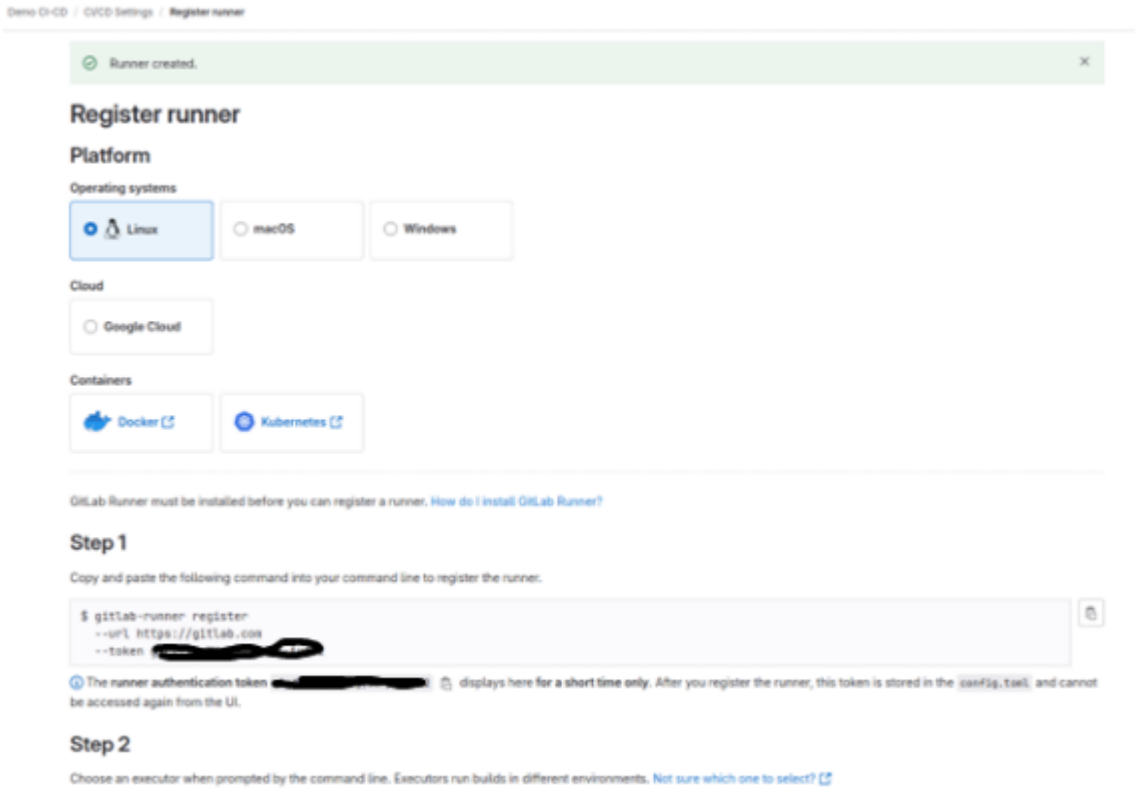



**Fig. 7** Screen 2 runner registration

- Copy the command of the Gitlab screen and add **sudo** before:

```
$ sudo gitlab-runner register --url ....
```

- Enter the Gitlab instance URL: press enter
- Enter a name for the runner: Enter the same of the runner description (*step 3, 3rd check*)
- Enter an executor: docker
- Enter default Docker image: openjdk:17

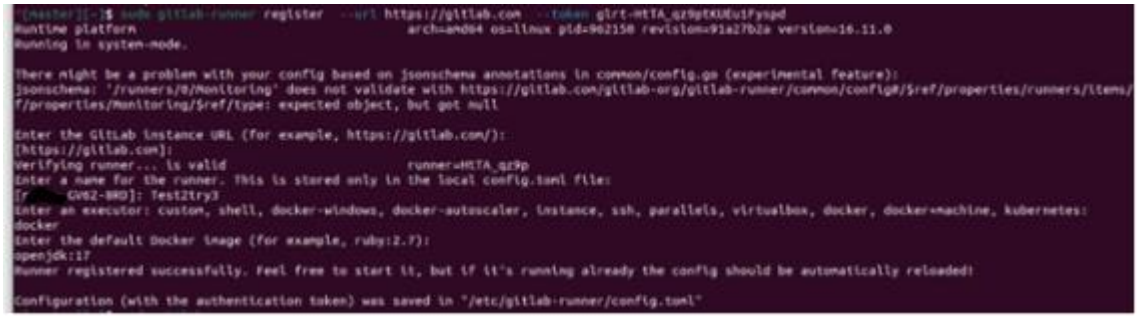



**Fig. 8** Example of each field in the process to register a runner

5. Check the runner is created:
   ```
   $ sudo gitlab-runner run
   ```

The runner should appear in green in the project runners section.

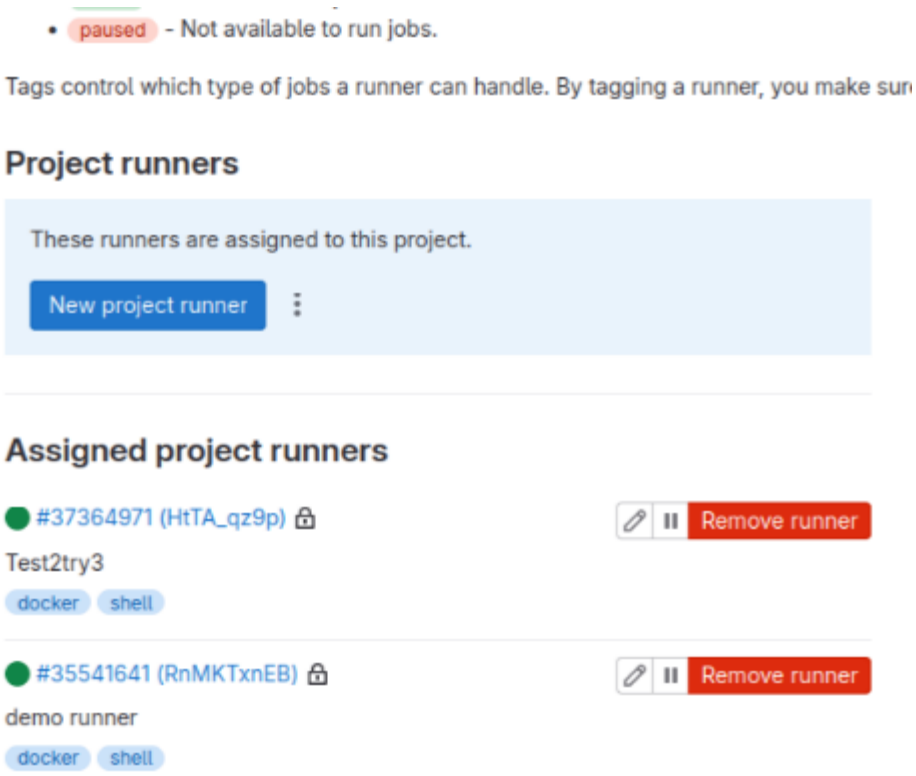



**Fig. 9** GitLab settings CI/CD screen with two runners correctly deployed.

Note: If the runner is with a black triangle with the message: *runner has never contacted this instance* run:

```
$ sudo gitlab-runner verify
```

### 7.6.2 Gitlab-ci.yml

The gitlab-ci file ensures that, every time someone commits and pushes the branch, the runner executes the sequence. The file is composed of 3 parts, the image, the definition of the stages and the stages.
The image is in which language/framework or environment you need to execute the stages.
The definition of the stages: it tells what each stage is called and its order of execution.
Stages: the ones in charge of carrying out the commands and/or scripts for the correct execution of each stage. Within a stage it is defined

- Stage: previously declared.
- Image: if it is needed for that stage.
- Script: where the commands and scripts for that stage are executed.

Besides, you can add extra functionalities to each stage such as: artifacts (to give the choice to download certain files) or allow_failure:true (which allows that stage to fail and continue executing the pipeline).

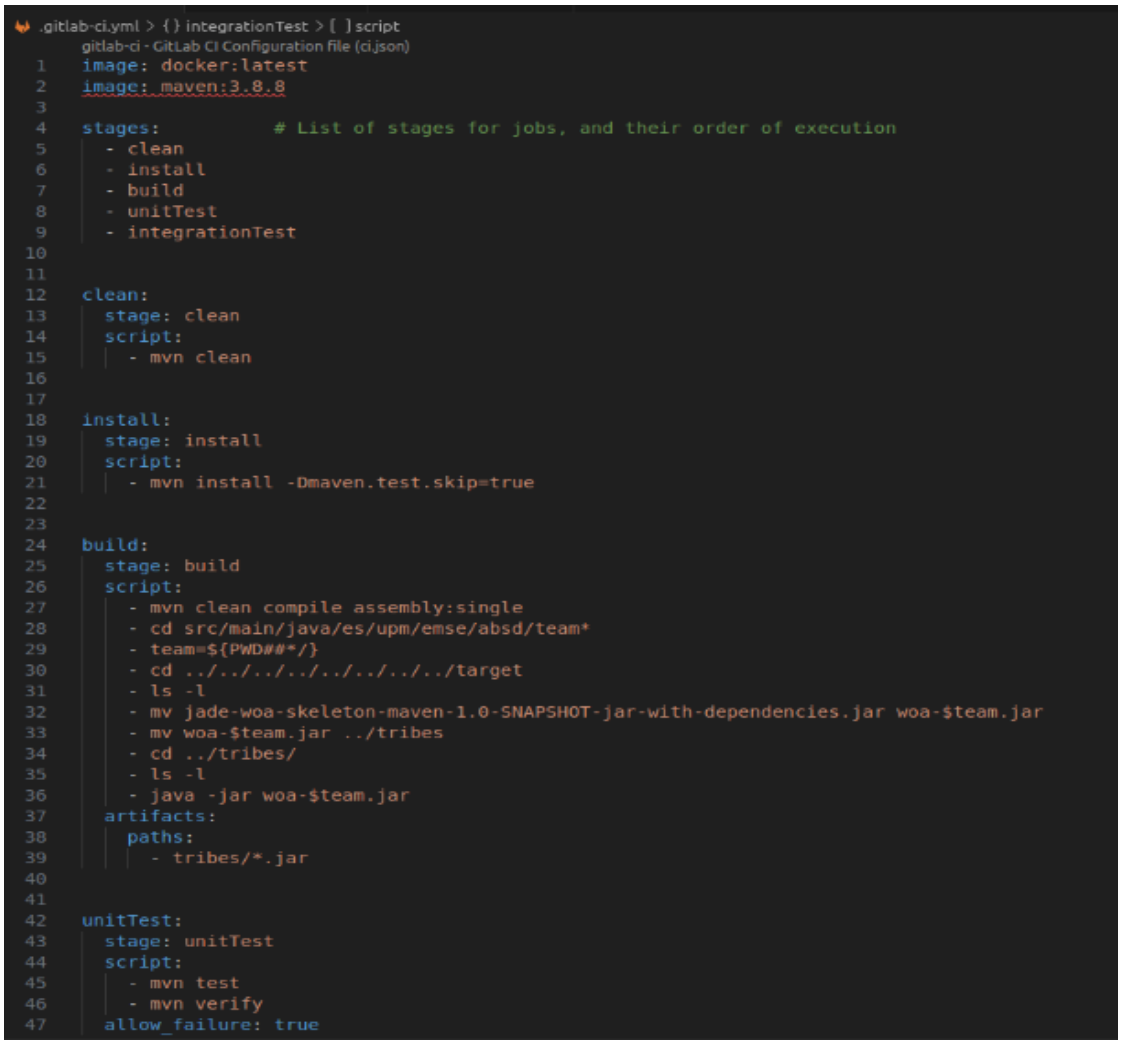



**Fig. 10** Section of the gitlab-ci.yml file

When a commit is uploaded to GitLab the pipeline looks like this:

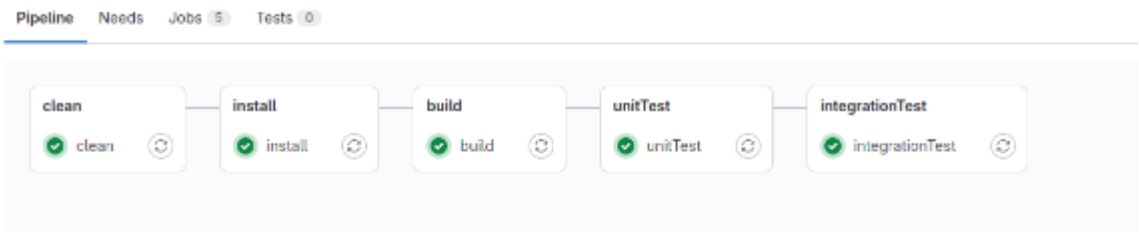



**Fig. 11** Pipeline of the project

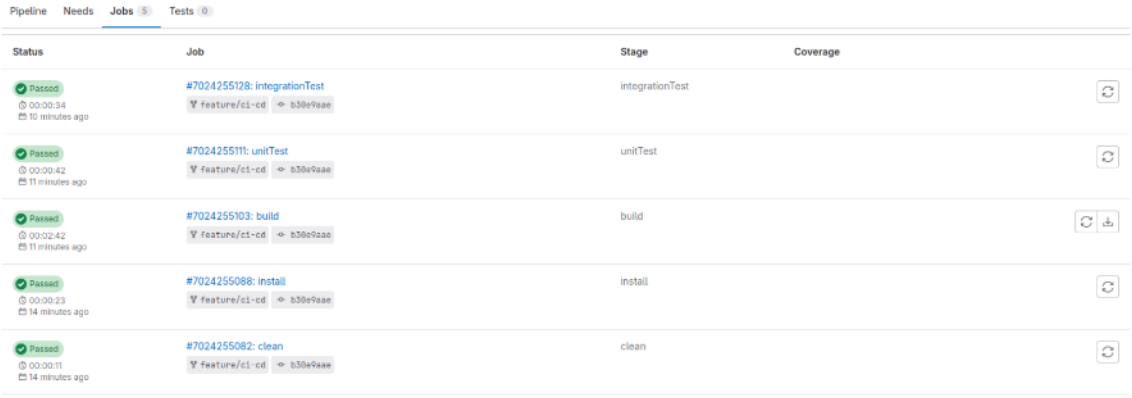



**Fig. 12** Pipeline of the project where we can see the possibility of downloading the artifact folder.

### 7.6.3 Test structure

All the necessary resources are in the **src.test** folder. To carry out the tests, we will rely on JUnit, which is a framework that marks the methods we choose as test methods. The test files must be named NameOfTheClass + Test, e.g., UtilTest.java.
JUnit has some annotations that will help us to identify the tests:

- **@Test**: Identify this method as a test.
- **@BeforeAll:** Used in the methods we will use for the configuration before all the tests.
- **@BeforeEach**: Like @BeforeAll but this method will be launched before each test, it is especially useful to reset the parameters before each execution.
- **@AfterAll:** It can be used to shut down the system or disable the agents at the end of the execution.

The tests of this skeleton are composed of three sections:

1. Unit tests, they check the operation of isolated functions and methods.
2. Ontology tests, they check that the structure of the ontology is correct.
3. Integration tests, check that the program runs correctly.

#### 7.6.3.1 Unit test

The unit tests oversee verifying that the main and auxiliary functions and methods fulfill their function. Since they should be developed as the practice progresses, some test cases have been generated as examples to guide the students.
For unit tests we have the following classes:

1. UtilTest: the UtilTest class has tests of the newMsg(), register() and genCID() methods and a small pre-initialization.
2. MainTest: The loading of a platform agent and a tribe agent is tested to ensure that no errors occur during the start of their deployment. At the end of these tests, the agent container is closed.
3. AgentTest: As there is no function or method inside the Agent class now, there is only the first structure.
4. AnswerBehaviour: has a small check that the salute() method returns information.

#### 7.6.3.2 Ontology test

The ontology tests oversee verifying that the ontology structure follows the agreements made by the students. For this purpose, some reflexive tests have been carried out so that, given some files with the agreements, it is verified that inside the folder src.main.java.es.upm.emse.absd.ontology they have the classes and vocabulary agreed upon.
Ontology tests use the following folders:

Ontology

Located at src.test.java.es.upm.emse.absd.ontology, hold one file named *OntologyTest.java*. Vocabulary, concept, action, and predicate tests are performed on the file, which check that the resources placed in the resource files exist in the ontology and that the vocabulary variables match with their contents.

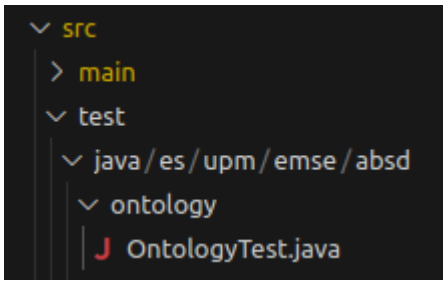



**Fig. 13** Path to the ontology test

utilTools

Located at src.test.java.es.upm.emse.absd.utilTools, hold 3 classes:

- **ClassFinder.java**: inside has 3 methods that return a String list of the search of all the classes that implement AgentAction, Concept or Predicate inside a given path
- **FileToList.java**: there is one method used in *OntologyTest.java* generates a list of Strings given the path to a *text* file.
- **SameName.java**: there are 2 methods. The first compares the variable with its content to see that they are named the same. The second converts snake_case to CamelCase.

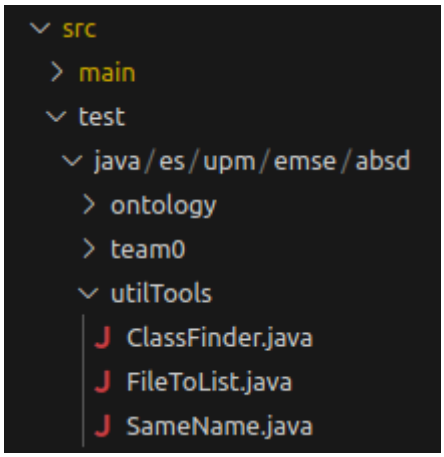



**Fig. 14** Path to the utilTools folder

Resources

Located at src.test.resources, holds 4 text files. Each of them has a list where the actions, concepts, predicates, and vocabularies agreed upon for the ontology should be placed. Each word of the list can be delimited with the following one by any delimiter either by line breaks, blank spaces, and/or commas.

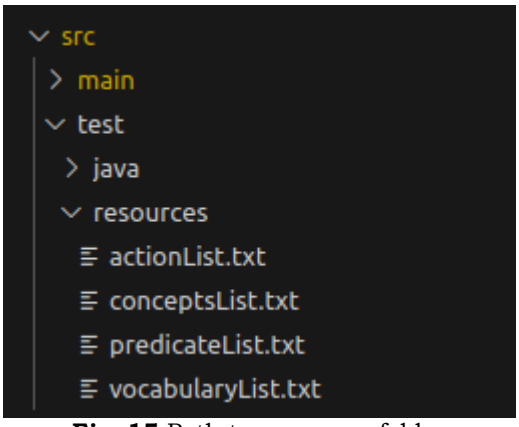



**Fig. 15** Path to resources folder

### 7.6.3.3 Integration test

The integration tests are included when executing the jar file. Two new commands have been added inside the main, -td or -tesdebug and -tb or -testbuild in which the environment is executed deploying the rma without the interface (since it would affect the pipeline and would not allow the test), the same as in the -d and -b cases, but after 10 seconds deployed, it closes automatically.

```
> java -jar woa-team0.jar -h
-------------------------------------
--- EMSE-ABSD JADE World of Agents ---
-------------------------------------
Agents:
    + Platform:
        - AgPlatform: An agent that responds when its called by its name.
    + Tribe:
        - AgTribe: An agent that try to talk with his/her colleague.
Usage: java -jar woa.jar [options]
    + options:
        -h, --help  Prints help
        -d, -debug  Run JADE agents from the compilation
        -b, -build  Run agents from generated jars
        -td, -testdebug  Run JADE agents from the compilation without interface and close the execution 10 seconds later
        -tb, -testbuild  Run agents from generated jars without interface and close the execution 10 seconds later
```

**Fig. 16** Response to the execution of help command in the .jar

These tests check that the environment is correctly deployed and that the agents in the system start communicating.

Este documento esta firmado por

| | | |
|---|---|---|
|  | **Firmante** | CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES |
| | **Fecha/Hora** | Sun Jun 30 20:44:59 CEST 2024 |
| | **Emisor del Certificado** | EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES |
| | **Numero de Serie** | 561 |
| | **Metodo** | urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature) |