Universidad de Cantabria

*Facultad*
*de*
*Ciencias*

# DESARROLLO DE SISTEMA DE FLUJOS DE TRABAJO EN LA NUBE PARA COMPUTACIÓN SOBRE CLIMA

(Development of cloud-based workflow system for climate computing)

Trabajo de Fin de Grado
para acceder al

**GRADO EN INGENIERÍA INFORMÁTICA**


**Autor: Marta Cue Trigos**

**Director: Héctor Pérez Tijero**

**Co-Director: Max Tuni San Martín**


**Junio - 2023**

# AGRADECIMIENTOS

Quiero expresar mi agradecimiento a todas las personas que me han apoyado en la realización de mi Trabajo de Fin de Grado.

En primer lugar, quiero agradecer a mi tutor en la empresa, Max, su guía y valiosos consejos a lo largo de todo el proceso. Sus conocimientos en este ámbito y su experiencia me han hecho aprender muchas cosas nuevas.

También quiero dar las gracias a todos los compañeros de Predictia. Tengo que agradecer su disposición para responder a mis preguntas y compartir sus conocimientos, lo cual facilitó enormemente mi trabajo.

Asimismo, quiero agradecer al director de mi TFG, Héctor, su dedicación y orientación a lo largo de todo el proceso. Sus sugerencias y comentarios fueron de gran ayuda para mejorar y perfeccionar mi trabajo.

A mis compañeros de clase, les agradezco su apoyo durante estos cuatro años. Hemos pasado muchas horas juntos haciendo prácticas y me llevo muy buenos recuerdos de todos ellos.

Por último, quiero expresar mi agradecimiento a mi familia. Agradezco su comprensión y paciencia para que pudiera dedicar tiempo a estudiar y a realizar este proyecto.

## Abstract

Bias adjustment is an important technical process in obtaining historical and future climate data for climate change impact and adaptation studies. In the company there is a bias adjustment web service connected to C3S (Copernicus Climate Change Service, where reliable information on the past, present and future climate situation of Europe and the rest of the world is made available to society), which allows users to apply various techniques on datasets.

Currently, this service uses a resource manager to assign tasks to computing resources and manage the data, which makes the process complex. Container technologies as well as cloud computing offer new ways of working with this type of data. This project aims to migrate the service to a Kubernetes cluster for more scalable and easier management.

**Keywords:** containers, Kubernetes, Argo Workflows, automation, cloud computing

## Resumen

El ajuste de sesgos es un proceso técnico importante en la obtención de datos climáticos históricos y futuros para estudios de impacto y adaptación al cambio climático. En la empresa existe un servicio web de ajuste de sesgos conectado al C3S (Servicio de cambio climático de Copernicus, donde se pone a disposición de la sociedad información fidedigna sobre la situación climática pasada, presente y futura de Europa y el resto del mundo), que permite a los usuarios aplicar diversas técnicas sobre conjuntos de datos.

Actualmente, este servicio utiliza un administrador de recursos para asignar tareas a recursos informáticos y gestionar los datos, lo que hace que el proceso sea complejo. Las tecnologías de contenedores así como la ejecución en la nube ofrecen nuevas formas de trabajar con este tipo de datos. Este trabajo tiene como objetivo la migración del servicio a un clúster Kubernetes para una gestión más escalable y fácil de administrar.

**Palabras clave:** contenedores, Kubernetes, Argo Workflows, automatización, computación en la nube

# Índice

# Índice de figuras



# Índice de tablas



# Índice de código

# 1. Introducción

## 1.1. Contexto

Los modelos climáticos globales y regionales son la principal fuente de información para apoyar y fundamentar los estudios de impacto y adaptación al cambio climático en diversos sectores. Estos estudios suelen requerir datos climáticos históricos y futuros (proyectados según diversos escenarios de emisiones) de alta resolución y no sesgados, que se obtienen habitualmente aplicando métodos de ajuste de sesgo a los datos en bruto. Sin embargo, a pesar de la simplicidad de la idea, el ajuste de sesgos (también denominado *bias adjustment*, BA) es una etapa muy técnica que implica un número de supuestos que no siempre se tienen en cuenta, como que los datos disponibles para cada región sean precisos y completos. Por lo tanto, las herramientas (y servicios) de BA deberían ofrecer un conjunto de métodos alternativos estándar para abarcar distintos escenarios.

El C3S[27] es uno de los principales proveedores autorizados de datos climáticos en todo el mundo. En particular, el Climate Data Store[13] (CDS) es un punto de acceso homogéneo para una gran variedad de conjuntos de datos climáticos, incluidas las proyecciones mundiales y regionales sobre el cambio climático.

Por otro lado, *ClimAdjust*[12] es un servicio web personalizado de ajuste de sesgos conectado al C3S, que cumple los anteriores requisitos. Es un servicio basado en la nube que permite a los usuarios aplicar estas técnicas sobre conjuntos de datos observacionales públicos y privados (propiedad del usuario para la aplicación concreta).

El ajuste de sesgos está formado por varias tareas como, por ejemplo, el preprocesamiento de datos o la agregación espacial (agrupación de datos espaciales en unidades más grandes o regiones geográficas). Estas tareas son planificadas y asignadas a recursos informáticos (máquinas virtuales) por un componente que actúa como administrador de recursos. El administrador de recursos, en este caso denominado *Java Resource Manager* (JRM), se encarga de planificar tareas específicas a partir de la definición de un experimento de ajuste de sesgos. Los recursos informáticos se organizan en colas, soportando diferentes máquinas virtuales (VMs) adscritas a cada cola. El componente interactúa con el proveedor de la nube, pudiendo asignar y configurar nuevas máquinas virtuales en las colas cuando sea necesario.

## 1.2. Motivación

El componente descrito previamente es bastante complejo, puesto que requiere gestionar numerosas máquinas virtuales, y a su vez debe ser capaz de crear y configurar nuevas máquinas virtuales. La solución actual de la empresa en la que se desarrolla este TFG se basa en que cada una de las máquinas virtuales de cómputo tiene un cliente JRM que se encarga de mantener los estados del nodo y lanzar localmente las tareas mediante contenedores Docker. Hoy en día, existen tecnologías que ofrecen ventajas respecto a la administración de contenedores a través de máquinas virtuales. Entre las tecnologías de gestión de contenedores más populares se encuentra Kubernetes, que es un orquestador de contenedores que permite gestionar y escalar aplicaciones en un entorno de producción.

Respecto a la solución basada en JRM, Kubernetes es más flexible y escalable, lo que significa que puede aumentar o disminuir el número de instancias de una aplicación según las necesidades de carga de trabajo. Ofrece una mayor disponibilidad, ya que puede detectar y recuperarse automáticamente de fallos en las aplicaciones o en los nodos del clúster. Finalmente, es más fácil de administrar, ya que proporciona herramientas y APIs para la gestión centralizada del clúster

y la automatización de tareas repetitivas. Esto justifica la necesidad de gestionar el servicio a través de un clúster Kubernetes, así como implementar un nuevo motor de cómputo encargado de ejecutar las tareas en Kubernetes.

## 1.3. Objetivos

El objetivo principal de este trabajo es desarrollar un nuevo motor de cómputo que se encargue de gestionar los flujos de trabajo (también conocidos como *workflows*) para realizar experimentos climáticos, así como desplegar el sistema en un clúster Kubernetes basado en *cloud* que soporte la ejecución de los mismos. Un flujo de trabajo es una secuencia de tareas predefinidas que deben realizarse para completar un proceso, mientras que un motor de cómputo en este contexto es un componente que gestiona la ejecución de los flujos de trabajo en un sistema.

Las tareas a realizar para conseguir este objetivo son:

- Implementar un componente de descarga de datos obtenidos del *cloud*.
- Desarrollo de un nuevo motor de cómputo que gestione solicitudes de creación de nuevos flujos de trabajo, planifique las diferentes tareas, gestione los volúmenes de datos y programe la ejecución de los flujos de trabajo.
- Desarrollo de una API para gestionar experimentos y flujos de trabajo.
- Validar la funcionalidad del sistema. Se realizarán pruebas unitarias para verificar el correcto funcionamiento de los componentes individuales, pruebas de integración para evaluar la interoperabilidad de los módulos y pruebas de sistema para garantizar el cumplimiento de los requisitos del sistema en su conjunto.
- Desplegar el sistema en un clúster Kubernetes.

# 2. Herramientas y tecnologías

La mayor parte del proyecto se ha desarrollado en el lenguaje de programación Java. El servicio de ajuste de sesgos es un servicio web, y Java es una de las mejores opciones para este tipo de aplicaciones web. Permite a los desarrolladores crear aplicaciones web que pueden manejar un gran número de solicitudes simultáneas y grandes volúmenes de datos y tiene un rico ecosistema de librerías y herramientas con amplia gama de funcionalidades.

## 2.1. GitLab

Un sistema de control de versiones de código es una herramienta que permite gestionar cambios en el código fuente de un proyecto software a lo largo del tiempo. Se ha utilizado el sistema de control de versiones Git, y GitLab como servidor de repositorios Git. Se ha usado el servidor interno de Gitlab de la empresa, creando los repositorios necesarios para los diferentes componentes desarrollados.

Además, se ha utilizado la metodología de desarrollo CI/CD (integración continua/entrega continua). Esta metodología busca automatizar y acelerar el proceso de entrega de software. GitLab Runner [18] es una herramienta que se utiliza para ejecutar y administrar trabajos de integración continua y entrega continua en GitLab. Los *runners* son servidores que ejecutan los trabajos de CI/CD en los repositorios de GitLab. Los *runners* de GitLab pueden ser instalados en diferentes sistemas operativos, y también pueden ser ejecutados en contenedores Docker. Nuestros trabajos de CI/CD los realizan *runners* que ya están configurados y registrados en el servidor de GitLab. GitLab Runner puede ser utilizado para ejecutar una amplia variedad de tareas, desde la compilación y prueba de código hasta la implementación y despliegue de aplicaciones.

## 2.2. Maven

Maven [6] es una herramienta de gestión de proyectos que se utiliza para gestión de dependencias. Sirve para automatizar tareas comunes como la compilación o el empaquetado. Maven utiliza un archivo de configuración llamado POM (Project Object Model) para describir el proyecto (dependencias en módulos y componentes externos, procesos, orden de compilación, plugins del propio Maven...). Este archivo permite a Maven descargar automáticamente las dependencias necesarias para el proyecto, lo que facilita la gestión de las mismas y evita la necesidad de almacenarlas en el repositorio de código.

## 2.3. Spring Boot

Spring Framework es un marco de desarrollo de aplicaciones Java que proporciona características y herramientas para crear aplicaciones escalables y modulares. Algunas de las características incluidas en Sping Framework son: inyección de dependencias, configuración basada en anotaciones o integración de bases de datos.

Spring Boot [26] es una tecnología para crear aplicaciones autocontenidas, basada en el marco de trabajo Spring. Proporciona una manera rápida y sencilla de crear aplicaciones de Spring sin tener que configurar manualmente todos los detalles subyacentes. Se encarga de las labores de configuración de dependencias y el despliegue del servicio de aplicaciones. Para ello, utiliza un servidor de aplicaciones embebido como Tomcat. Spring Boot proporciona un conjunto completo de herramientas para la creación y el desarrollo de aplicaciones, incluidas herramientas para la generación de código y la configuración automática de las propiedades. Al utilizarse con Maven, se crean aplicaciones más rápido y de forma más eficiente. Esta tecnología servirá en este proyecto para desarrollar el servidor web, inyectar dependencias o gestionar la configuración de manera más sencilla.

## 2.4. Docker

Docker es una plataforma de software que permite crear, probar e implementar aplicaciones rápidamente [1]. Docker empaqueta software en contenedores, que incluyen todo lo necesario para que el software se ejecute, incluidas bibliotecas, herramientas de sistema y código. Docker es un sistema operativo para contenedores. De manera similar a cómo una máquina virtual virtualiza (elimina la necesidad de administrar directamente) el hardware del servidor, los contenedores virtualizan el sistema operativo de un servidor. Docker se instala en cada servidor y proporciona comandos sencillos que puede utilizar para crear, iniciar o detener contenedores[17].

Las imágenes Docker son archivos inmutables que pueden ser utilizados para crear contenedores Docker. Un contenedor Docker es una instancia en ejecución de una imagen. Docker dispone de un servicio de almacenamiento de imágenes en la nube, llamado Docker Hub[16]. Permite a los desarrolladores almacenar, descargar y compartir imágenes de manera fácil y segura. Este servicio ofrece una amplia variedad de imágenes de software preconstruidas, que se pueden utilizar como base para las aplicaciones propias.

Por otro lado, Docker puede construir imágenes leyendo imágenes de un Dockerfile. Un Dockerfile es un archivo de texto que contiene los comandos para construir una imagen. Un Dockerfile define todas las dependencias, configuraciones y recursos necesarios para ejecutar una aplicación en un contenedor. Los comandos en un Dockerfile incluyen información sobre la imagen base a usar, cómo copiar archivos en el contenedor, qué comandos ejecutar al iniciar el contenedor, etc. En la documentación oficial de Dockerfile (`https://docs.docker.com/engine/reference/builder/`) se obtiene información sobre el funcionamiento de las variables de entorno, las principales directivas, los volúmenes y la exposición de puertos.

Para el proyecto, se usará Docker para empaquetar las aplicaciones, así como para utilizar otras imágenes del registro Docker Hub y el registro de imágenes propio de la empresa.

## 2.5. Kubernetes

Kubernetes [2] es una plataforma portable y extensible de código abierto para administrar cargas de trabajo y servicios. Kubernetes facilita la automatización y la configuración declarativa. Tiene un ecosistema grande y en rápido crecimiento. El soporte, las herramientas y los servicios están ampliamente disponibles. Kubernetes ofrece un entorno de administración centrado en contenedores. Kubernetes orquesta la infraestructura de cómputo, redes y almacenamiento para que las cargas de trabajo de los usuarios no tengan que hacerlo.

De forma general, Kubernetes está formado por los siguientes componentes clave[14]:

- **Nodos**: son los servidores que albergan los contenedores.
- **Clúster**: es un grupo de nodos que trabajan juntos para ejecutar aplicaciones.
- **Recursos**: son los objetos que describen las aplicaciones y sus requisitos, como despliegues, servicios, configuraciones, etc.
- **Controladores**: son los componentes de Kubernetes que supervisan el estado de los recursos y aseguran que se cumplan las especificaciones descritas en ellos.
- **Namespaces**: Mecanismo para aislar grupos de recursos en un solo clúster

Además, Kubernetes dispone de una herramienta de línea de comandos, `kubectl`, que permite interactuar con un clúster de kubernetes. Permite crear, actualizar y desplegar aplicaciones, supervisar el estado de los recursos y monitorizar el rendimiento de las aplicaciones en tiempo

real. Para obtener una lista completa de las operaciones disponibles se puede consultar la documentación oficial (https://kubernetes.io/docs/reference/kubectl/).

Por otro lado están los objetos de Kubernetes. Son entidades persistentes que se utilizan para representar el estado del clúster. También representan la configuración deseada de una aplicación en un clúster. Hay varios tipos de objetos, incluyendo:

- **Pod**: Una unidad básica de trabajo en Kubernetes. Es un grupo de uno o más contenedores, con almacenamiento compartido y recursos de red, y una especificación sobre cómo ejecutar los contenedores.
- **Deployment**: Objeto que se encarga de administrar y mantener una versión deseada de una aplicación en un clúster. Asegura que los pods que ejecutan la aplicación estén disponibles y en número adecuado. Si los pods fallan, se crearán nuevos para mantener la disponibilidad. En resumen, facilitan la gestión y despliegue de las aplicaciones.
- **Service**: Objeto que describe cómo se accede a las aplicaciones, tal como un conjunto de pods, y que puede descubrir puertos y balanceadores de carga. Kubernetes otorga a los pods su propia dirección IP y un nombre DNS para un conjunto de pods, y puede balancear la carga entre ellos.
- **Volume**: Medio de almacenamiento compartido para los pods. Son útiles para compartir datos entre contenedores o mantenerlos persistentes después de que los pods mueran o se reinicien. Los volúmenes pueden ser de diferentes tipos, como una carpeta de un sistema de archivos, una unidad de almacenamiento en línea, o un volumen de almacenamiento en bloques (por ejemplo, un disco duro o SSD).
- **Secret**: Almacena información sensible como contraseñas, credenciales, tokens, claves SSH u otros datos confidenciales que no deben exponerse a usuarios o aplicaciones.
- **ConfigMap**: Almacena información de configuración de la aplicación en formato clave-valor. Se suelen utilizar para compartir información de configuración entre servicios. Proporciona a las aplicaciones y servicios que se ejecutan en los pods un conjunto de credenciales para autenticarse con el API Server de Kubernetes y acceder a otros recursos del clúster.
- **Service Account**: Representa una entidad que se utiliza para proporcionar a las aplicaciones y servicios un conjunto de credenciales para autenticarse con el API Server de Kubernetes y acceder a otros recursos del clúster.
- **Role**: Un rol contiene reglas que representan un conjunto de permisos. Un rol establece permisos para un namespace particular. Se establecen permisos para la creación o eliminación de pods.
- **Role Binding**: Un *role binding* otorga los permisos definidos en un rol a un usuario o conjunto de usuarios. Contiene una lista de sujetos (usuarios, grupos o cuentas de servicio) y una referencia al rol que se otorga. Un RoleBinding otorga permisos dentro de un namespace.
- **Priority Class**: Es un objeto sin namespace que define un nombre de clase de prioridad a su valor entero de prioridad. Cuanto mayor sea este valor, mayor será la prioridad. Una vez que exista una o más *priority class*, se pueden crear pods con una clase de prioridad.

Cabe destacar que estos recursos se pueden definir en formato YAML. Kubernetes se usará como orquestador de contenedores, facilitando la gestión de la infraestructura de manera eficiente y escalable. En Kubernetes existen otro tipo de objetos, pero en el proyecto los objetos utilizados son los descritos en este documento.

## 2.6. Argo Workflows

Argo Workflows [8] es un motor de flujos de trabajo de código abierto nativo de contenedores para orquestar trabajos paralelos en Kubernetes. Argo Workflows está implementado como un CRD (Custom Resource Definition). Como resultado, los flujos de trabajo de Argo pueden gestionarse a través de la línea de comandos con el comando `kubectl` y se integran de forma nativa con otros servicios de Kubernetes, como *volumes* o *secrets*. Argo Workflows ofrece las siguientes funcionalidades:

- Definir flujos de trabajo en los que cada paso es un contenedor.
- Modelar flujos de trabajo de varios pasos como una secuencia de tareas o con dependencias entre tareas mediante un gráfico dirigido acíclico (DAG).
- Ejecutar fácilmente trabajos de cálculo intensivo para el aprendizaje automático o el procesamiento de datos.
- Ejecutar pipelines CI/CD de forma nativa en Kubernetes sin configurar productos de desarrollo de software complejos.

Los flujos de trabajo en Argo se definen utilizando archivos YAML o JSON. También proporciona una interfaz de usuario web que permite a los usuarios visualizar y monitorizar el progreso de los flujos de trabajo en tiempo real. Además, también permite la integración con sistemas de notificación y alerta para informar a los usuarios sobre los resultados de los flujos de trabajo. Argo Workflows será de utilidad en este proyecto, puesto que para realizar los experimentos del clima será necesario ejecutar trabajos paralelos en contenedores Docker.

## 2.7. Helm

**Helm**[19] es un administrador de paquetes para Kubernetes que simplifica la implementación y administración de aplicaciones. Al usar Helm, puedes empaquetar todos los recursos Kubernetes necesarios para ejecutar una aplicación en un solo *chart*, lo que facilita la implementación y administración de aplicaciones en diferentes entornos.

Un *Helm chart* es un paquete que contiene todos los recursos Kubernetes necesarios para ejecutar una aplicación, incluidos los *deployments*, *services*, *configmaps*, etc. El *chart* está organizado en una estructura de directorios que incluye algunos archivos:

- `Chart.yaml`: Contiene los metadatos del *chart*, como el nombre, la versión y la descripción.
- `values.yaml`: Contiene los valores de configuración predeterminados para el *chart*.
- `templates/`: Contiene las definiciones de recursos de Kubernetes para el *chart*, escritas en YAML o JSON.

En cuanto a su arquitectura, Helm consta de dos componentes: la CLI de Helm y *Tiller*. La CLI de helm es una herramienta de línea de comandos que se utiliza para interactuar con Helm y Tiller es un componente del lado del servidor que es responsable de implementar y administrar los recursos en el clúster de Kubernetes.

La aplicación a desplegar requiere múltiples recursos Kubernetes, como *volumes*, *secrets*, *deployments*, etc. Sin Helm habría que crear manualmente cada uno de los recursos, lo que puede llevar mucho tiempo y ser propenso a errores. Además, los *charts* de Helm se pueden compartir y reutilizar (por ejemplo, en el repositorio comunitario **ArtifactHub**[10]), lo que facilita la implementación de la misma aplicación en varios clústeres. En este sentido, se pretende reutilizar en la medida de lo posible los *charts* ya existentes en la empresa. El despliegue se hará ejecutando un solo comando a través de la línea de comandos.

## 2.8. Formatos de serialización de datos: YAML y JSON

YAML (*YAML Ain't Markup Language*) es un formato de serialización de datos legible por humanos y orientado a objetos. Es frecuentemente utilizado para la configuración de aplicaciones y para el intercambio de datos entre sistemas que utilizan diferentes lenguajes de programación. Su sintaxis se basa en la indentación (uso de espacios en blanco para definir la estructura del documento) y en el uso de marcadores para definir los elementos de datos. YAML también tiene la capacidad de definir tipos de datos complejos, como listas y diccionarios, y admite referencias entre ellos.

JSON (JavaScript Object Notation) es un formato de texto ligero utilizado para el intercambio de datos. Se basa en dos estructuras de datos principales: objetos y matrices. Un objeto es una colección no ordenada de pares de clave-valor, y una matriz es una lista ordenada de valores. Tanto los objetos como las matrices se pueden anidar para formar estructuras más complejas.

YAML se utiliza para configurar las aplicaciones que realizan cada una de las tareas de cada flujo de trabajo. Por otra parte, para definir los flujos de trabajo en Argo y para obtener la configuración de los experimentos climáticos usamos JSON.

## 2.9. Servicio de almacenamiento en la nube Amazon S3

Amazon S3 (Simple Storage Service) [5] es un servicio de almacenamiento en la nube ofrecido por Amazon Web Services (AWS) que permite almacenar y recuperar grandes cantidades de datos en cualquier momento y desde cualquier lugar de la web.

Ofrece una API para interactuar con el servicio de almacenamiento en la nube utilizando diferentes lenguajes de programación. Con esta API, se pueden crear, obtener y eliminar objetos de almacenamiento en la nube, así como también listar objetos y realizar operaciones de control de acceso y políticas de almacenamiento.

Los datos se almacenan en "buckets"(cubos), que son contenedores virtuales para almacenar objetos (archivos) en la nube. Los buckets tienen un nombre que debe ser único en todo el sistema S3. Cada objeto almacenado en S3 tiene una clave única que actúa como identificador del objeto dentro del bucket. Los objetos pueden ser de cualquier tipo de archivo, como imágenes, videos, documentos, etc.

# 3. Antecedentes: descripción del servicio original

En esta sección se hace una descripción general del servicio, indicando aspectos científicos y la arquitectura del sistema. La descripción incluida en este apartado es el punto de partida del proyecto, sobre el que se va a construir la solución deseada.

La solución existente expone un servicio web para aplicar técnicas de ajuste de sesgos definidas por el usuario a las proyecciones climáticas globales y regionales publicadas en el *Climate Data Store*.

Las técnicas de ajuste de sesgos utilizan dos *datasets* diferentes:

- Salidas del modelo: el conjunto de datos que será ajustado. Las proyecciones climáticas consideran tanto escenarios históricos como futuros.
- Referencia observacional: el conjunto de datos cuadriculado o basado en puntos que se utiliza para ajustar los resultados del modelo, generalmente calibrando algunas estadísticas de la distribución del modelo hacia la distribución observada.

En los estudios de impacto y adaptación al cambio climático, variables como la temperatura y la precipitación son esenciales para comprender y evaluar los impactos del cambio climático en diferentes aspectos de la sociedad. Con estas variables se realizan ajustes que ayudan a mejorar la precisión de los modelos climáticos y las proyecciones futuras. Las variables consideradas en este servicio son: temperatura, precipitación, humedad y velocidad del viento.

El ajuste de sesgos se realiza usando *climate4R*, un framework comunitario implementado con lenguaje R para el acceso transparente a los datos climáticos y el postprocesamiento. Utiliza distintas estructuras para manejar de manera eficiente los datos de las observaciones y las proyecciones climáticas. El ajuste de sesgos lo realiza el paquete *downscaleR*, que incluye diversas técnicas de ajuste de sesgos.

## 3.1. Descripción de un experimento climático

Tal y como se ilustra en la figura 1, un experimento de ajuste de sesgos está formado por varias etapas, cada una de ellas de un tipo y cada una tiene unas dependencias con otras etapas. Estas etapas son: recuperación de datos, ajuste de sesgos y posprocesado de datos.

1. **Catalog Datasource**: Se trata de la etapa inicial y sirve para la recuperación de datos del catálogo. Para este caso se especifican el *dataset* (el conjunto específico de datos climáticos del CDS que se ha escogido para realizar el experimento), variables a ajustar, la cobertura espacial (áreas geográficas que están incluidas en el estudio) y el tipo de máscara (técnica utilizada para delimitar áreas específicas de interés en un conjunto de datos climáticos. La máscara se aplica asignando valores binarios a cada ubicación en función de si se encuentra dentro o fuera del área de interés). Además, para esta tarea existen tres subetapas:
    - **Projection**: Esta subetapa está relacionada con escenarios futuros. Se tienen que especificar los modelos utilizados (representaciones matemáticas de los procesos físicos que gobiernan el clima, desarrollados por distintas instituciones) y los escenarios de emisión futuros (políticas climáticas respecto a la emisión de gases de efecto invernadero). Los escenarios tienen un nombre, fecha de inicio y fecha de fin. Hay que tener en cuenta que cada tipo de *dataset* dispone de datos de unos intervalos de tiempo concretos.
    - **Calibration**: Esta subetapa se refiere a evaluaciones de los modelos hacia el pasado. Tiene los mismos modelos que *projection*, las mismas variables y los escenarios son históricos.

- **Reference**: Esta subetapa está relacionada con los datos de observaciones, basados en datos medidos en diferentes estaciones, por lo cual no hay escenarios ni modelos. Las fechas de inicio y fin tienen que coincidir con las fechas de *calibration* para poder realizar los ajustes de sesgo.

2. **Custom Datasource**: Se trata de una etapa opcional de descarga. Surge si en *reference* se quieren usar datos privados del usuario, en vez de utilizar los datos proporcionados en los *datasets* existentes. El fichero proporcionado por el usuario está disponible en el S3, simplemente habrá que realizar la descarga del mismo.

3. **Bias Adjustment**; En esta etapa ya se realiza el ajuste de sesgos. Es necesario especificar la técnica a utilizar, el tipo de validación (proceso de comparación de los resultados del modelo con datos de referencia para determinar si los modelos reproducen adecuadamente los comportamientos observados en la realidad) y sus dependencias son las subetapas de *Catalog Datasource* y *Custom Datasource*.

4. **Post Processing**: Esta es la última etapa. Aquí se produce el formato de salida final (por ejemplo, NetCDF, Excel o archivos de texto) con el resultado del experimento.

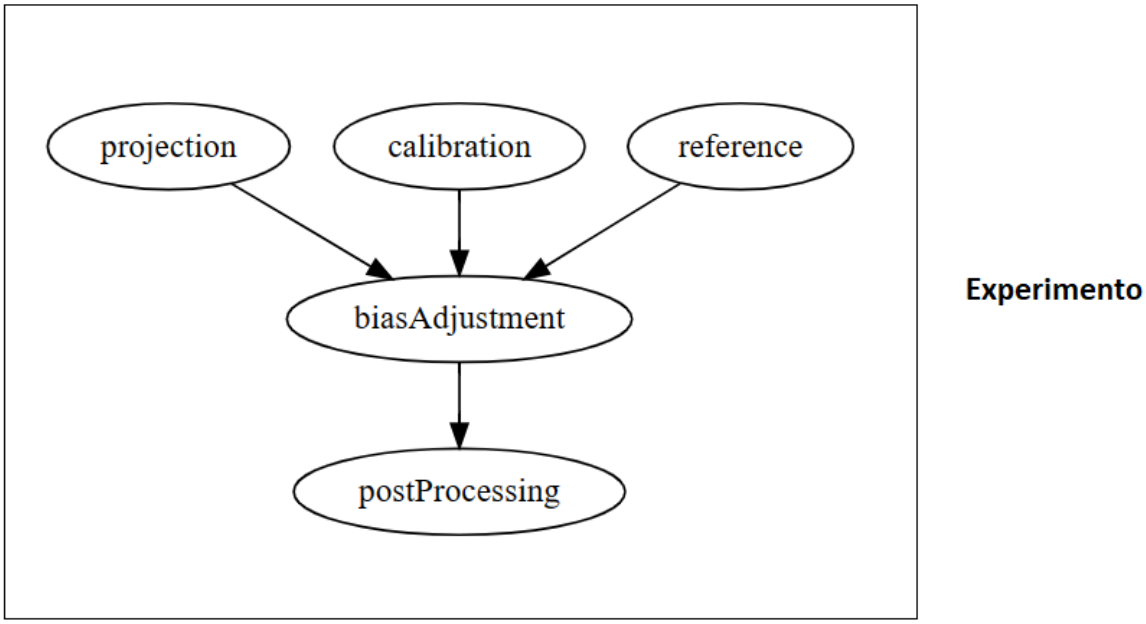


Figura 1: Esquema de un experimento de ajuste de sesgos

## 3.2. Arquitectura del sistema

La figura 2 muestra la arquitectura del sistema a alto nivel. Se presentan los principales componentes como el subsistema de almacenamiento o el subsistema de cómputo, y las interacciones entre ellos.

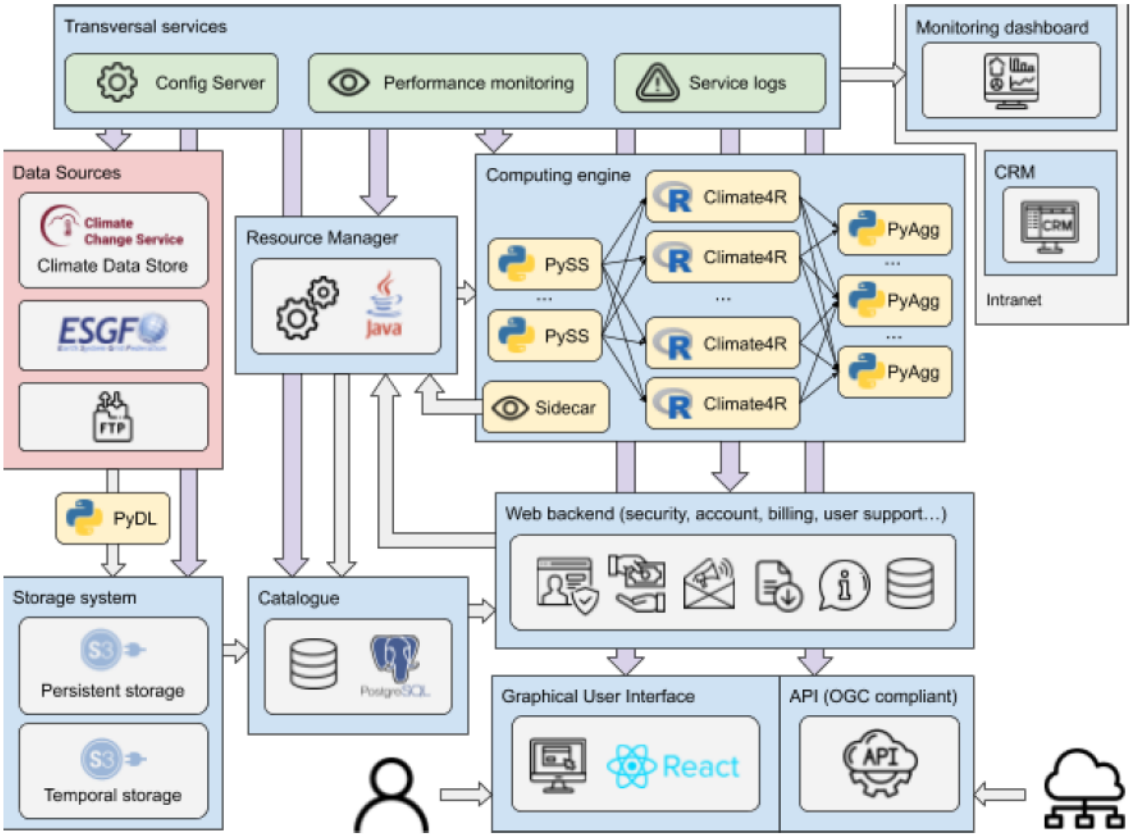


Figura 2: Diagrama de arquitectura de alto nivel. Fuente: Predictia Intelligent Data Solutions

### Subsistema de almacenamiento de datos

Este subsistema está desplegado en la nube y almacena datos de observación privados de los usuarios y los resultados de los ajustes de sesgo. En la figura 2 este subsistema está representado con el nombre de *storage system* y tiene las siguientes características:

- Sistema paralelizable: para mejorar el rendimiento, los conjuntos de datos están fragmentados espacio-temporalmente y se accede a ellos en paralelo. El almacenamiento admite una gran cantidad de lecturas/escrituras simultáneas sin comprometer el rendimiento.
- Escalable: el sistema debe admitir una gran cantidad de elementos atómicos mientras mantiene tiempos de lectura/escritura predecibles.
- Tolerancia a fallos: Los datos se almacenan en varias unidades en diferentes máquinas, evitando pérdida de datos en caso de fallos de hardware. El sistema de almacenamiento permanece en línea durante el reemplazo de las unidades defectuosas.
- Conectividad con servicios de cómputo y capa de visualización: el sistema de almacenamiento es accesible con baja latencia desde los diferentes componentes de la arquitectura, con una API HTTP.

### Subsistema de administración de recursos

Es un componente (representado en el diagrama con el nombre *resource manager*) que se encarga de la planificación de trabajos. Su función es planificar tareas a partir la definición de un experimento de BA, y asignarlas a los recursos informáticos (VMs).

Las distintas tareas de un trabajo componen un grafo dirigido, en el que algunas tareas dependerán de la de otras, mientras que otras podrán ejecutarse de forma independiente. El componente enviará las tareas a las colas tan pronto como haya capacidad libre en la cola, teniendo en cuenta las prioridades de las tareas.

#### Subsistema de cómputo

El motor de cómputo está desplegado en un clúster de máquinas virtuales en el proveedor de la nube. En la figura 2 está representado con el nombre *computing engine*. Además, para reducir tiempos de procesamiento, numerosas tareas de computación son definidas por el JRM y corren en paralelo.

El motor de cómputo que soporta la ejecución de las tareas tiene las siguientes características:

- Tamaño configurable de las máquinas virtuales: Diferentes tipos de usuarios pueden requerir diferentes enfoques en términos de velocidad y coste. Se definen prioridades y algunas tareas disponen de recursos informáticos adicionales.
- Tamaño dinámico: El número de recursos informáticos variará dinámicamente, añadiendo más nodos concurrentes para mantener el nivel de calidad de servicio en periodos de alta demanda.
- Conectividad rápida con el almacenamiento de datos, ya que se transfieren grandes cantidades de datos desde y hacia el sistema de almacenamiento.

El proveedor de cloud soporta el aprovisionamiento programático de diferentes tipos de máquinas virtuales a través de una API HTTP.

Las tareas son realizadas por contenedores, descritos a continuación:

- **Python Spatial Selector (PySS)**: Lee la configuración de la tarea, prepara los datos para una región determinada a partir de los diferentes fragmentos (disponibles localmente en el nodo informático) y convertirlos al formato de entrada que necesita Climate4R.
- **Climate4RWrapper**: Implementa un método automático de fragmentación espacial que, en función de los recursos informáticos disponibles (por ejemplo, memoria RAM), elige un tamaño de fragmentación óptimo para evitar errores de memoria que no degraden el rendimiento y adaptar el formato de datos original al utilizado por Climate4R[20]. Recibe algunos archivos NetCDF fragmentados espacialmente de PySS y produce un archivo NetCDF equivalente con ajuste de sesgo.
- **Python Aggregator (PyAgg)**: Gestiona los datos proporcionados por el climate4R. Su primera tarea será fusionar los datos fragmentados espacialmente en un solo archivo. También realiza varias operaciones de reducción de dimensiones, que dependen de las demandas del usuario. Estas operaciones pueden incluir agregaciones espaciales en áreas específicas (polígonos), así como reducción de muestreo temporal.

#### Subsistema de datos: catálogo

El catálogo (representado en la figura 2 con el nombre *catalogue*) mantiene el estado de los *datasets*. Expone una interfaz REST para permitir que otros componentes consulten, guarden y actualicen los *datasets* de manera unificada. La unidad de trabajo del catálogo es el *chunk* (compuesto por metadatos como cobertura espacial, variables, ubicación del archivo, etc.) almacenado en PostgreSQL como base de datos y el archivo de datos (archivo NetCDF) guardado en un almacén de objetos compatible con S3. Proporciona a los procesos un lugar centralizado donde consultar los datos necesarios.

**Subsistema de gestión de usuarios**

Componente escrito en Java que implementa distintos servicios de *backend*. En la figura 2 se representa con el nombre *web backend* y proporciona funcionalidades como el registro de usuarios, gestión de cuentas o asistencia al usuario.

# 4. Análisis y especificación de requisitos

Esta sección describe la fase de análisis de requisitos y posteriormente, detalla los requisitos funcionales y no funcionales que debe cumplir el sistema.

## 4.1. Análisis de requisitos

En la fase de análisis del proyecto, se han recogido distintos requisitos que se deben cumplir para poder afirmar que el motor de cómputo genera el resultado correcto y cumple su función adecuadamente. Estos requisitos han sido obtenidos a través de un análisis personal de las necesidades del sistema, así como de entrevistas con los responsables y desarrolladores del servicio.

## 4.2. Requisitos funcionales

Los requisitos funcionales son las declaraciones de cómo debe comportarse un sistema. Define lo que un sistema debe hacer para satisfacer las necesidades o expectativas del usuario. La tabla 1 recoge este tipo de requisitos.

Tabla 1: Requisitos funcionales

| ID | Descripción |
|---|---|
| RF01 | El sistema puede recibir peticiones de ejecución de experimentos climáticos en formato JSON |
| RF02 | El sistema, a partir de la definición del experimento, planificará diversas tareas a realizar por contenedores |
| RF03 | El sistema consultará el catálogo para poder descargar los datasets requeridos |
| RF04 | El sistema generará los ficheros de parámetros para cada tarea |
| RF05 | El sistema definirá un grafo dirigido acíclico (DAG) para establecer las dependencias entre tareas |
| RF06 | El sistema devolverá como salida 1 o 2 ficheros en el formato escogido en la definición del experimento (NetCDF, Excel, formato de texto) |
| RF07 | El sistema permitirá establecer prioridades en la ejecución de los experimentos |
| RF08 | El sistema permitirá consultar el estado de un *workflow* (estado, uso de CPU, resumen de tareas, etc.) |
| RF09 | El sistema permitirá consultar el estado de las tareas de un *workflow* |
| RF10 | El sistema permitirá volver a ejecutar las tareas fallidas de un *workflow* |
| RF11 | El sistema permitirá eliminar un *workflow*, incluyendo la eliminación de las tareas en ejecución |

## 4.3. Requisitos no funcionales

Los requisitos no funcionales son aquellos que definen características no directamente relacionadas con la funcionalidad del sistema, pero que son importantes para garantizar su calidad,

eficiencia y seguridad. A veces se conocen como restricciones o requisitos de calidad. En la tabla 2 se presentan los requisitos no funcionales.

Tabla 2: Requisitos no funcionales

| ID | TIPO | DESCRIPCIÓN |
|---|---|---|
| RNF01 | Seguridad | El sistema empleará *secrets* de Kubernetes para gestionar datos confidenciales |
| RNF02 | Monitorización | El sistema debe contar con herramientas de monitorización y análisis para supervisar el rendimiento y la actividad del workflow |
| RNF03 | Monitorización | El sistema debe tener un *log* que registre las acciones realizadas por las diferentes tareas |
| RNF04 | Escalabilidad | El sistema debe ser capaz de escalar horizontalmente de forma proporcional al número de tareas que se ejecutan en el sistema, según la disponibilidad de recursos del clúster |
| RNF05 | Rendimiento | El sistema debe tener la capacidad de procesar grandes cantidades de datos y realizar cálculos complejos durante la ejecución |
| RNF06 | Rendimiento | Cada contenedor encargado de realizar una tarea tendrá requerimientos y límites de recursos de CPU y memoria |
| RNF07 | Rendimiento | El sistema debe ser capaz de lanzar los workflows de manera eficiente y asegurando una respuesta rápida, entre uno y diez segundos |
| RNF08 | Disponibilidad | El sistema debe estar disponible para la planificación de experimentos, aunque haya nodos que no están disponibles |
| RNF09 | Resiliencia | El sistema debe ser capaz de recuperarse rápidamente en caso de errores o interrupciones en la ejecución de los *workflows* |
| RNF10 | Integración | El sistema debe ser fácil de integrar con otras herramientas y servicios |

# 5. Diseño del sistema

En esta sección se describe primero el diseño arquitectónico del sistema propuesto para este trabajo de fin de grado, entrando después en los detalles del diseño.

## 5.1. Diseño arquitectónico

La figura 3 muestra el nuevo diagrama de arquitectura del sistema en alto nivel. Se mantiene la arquitectura ya existente, sustituyendo el componente de administración de recursos por el nuevo motor de cómputo (representado en la figura 3 por *Argo Runner*), y el motor de cómputo anterior por el servidor de Argo Workflows.

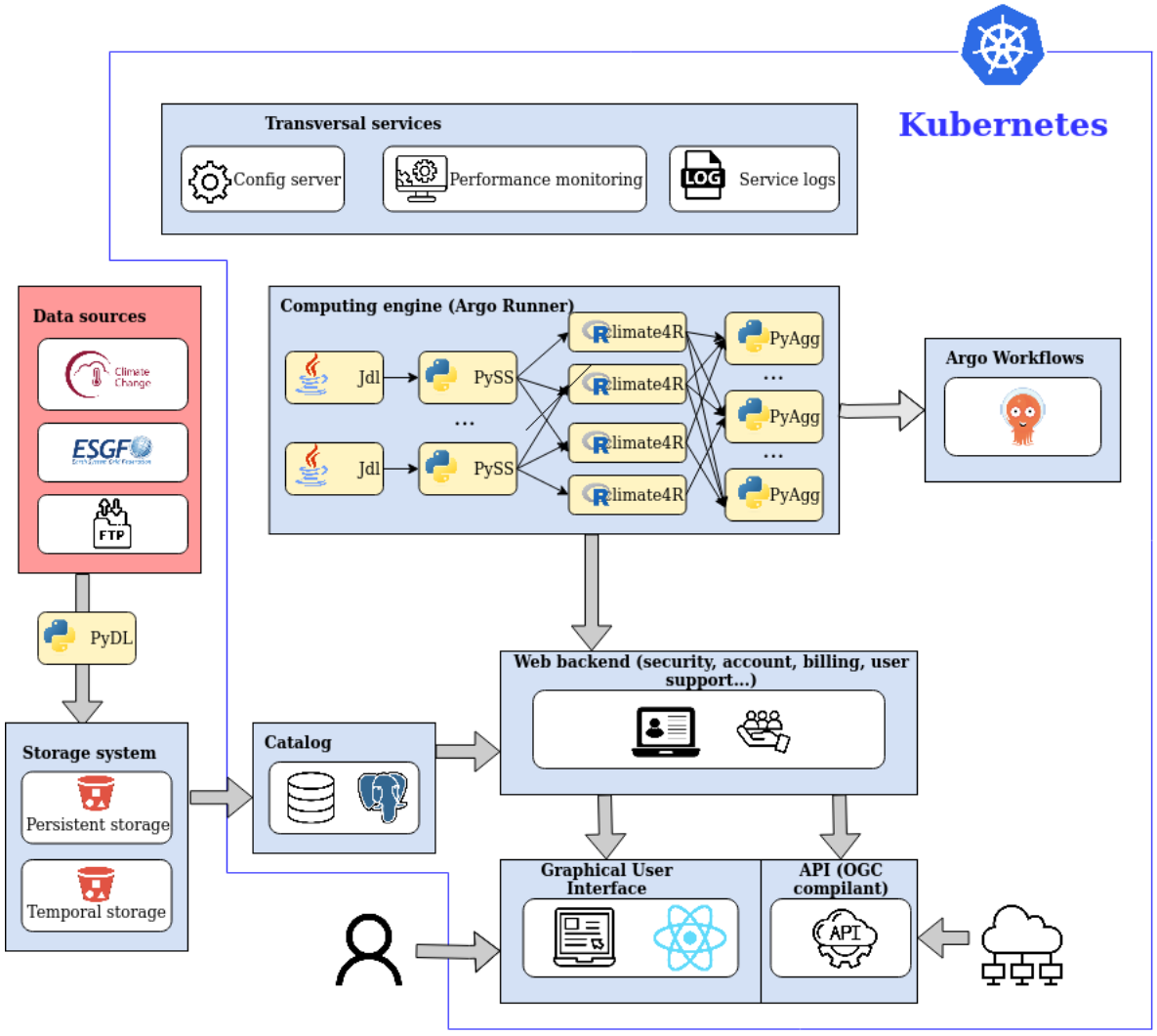


Figura 3: Arquitectura del sistema propuesto

Los cambios realizados a alto nivel solo implican la sustitución de dos componentes. Sin embargo, a bajo nivel se producen cambios significativos debido a la migración del sistema a Kubernetes. Al pasar de una solución basada en máquinas virtuales a una arquitectura basada en Kubernetes, se producen los siguientes cambios en la arquitectura del sistema a bajo nivel:

1. Orquestación y administración centralizada: Kubernetes orquesta los servicios en el clúster. Proporciona un plano de control centralizado que gestiona la programación, escalado, despliegue y monitorización de los contenedores en diferentes nodos del clúster

2. Abstracción del hardware subyacente: En lugar de interactuar directamente con máquinas virtuales, los desarrolladores y administradores de sistemas interactúan con los recursos de Kubernetes

3. Gestión declarativa: la configuración de las aplicaciones se definen de forma declarativa mediante archivos de configuración YAML. Kubernetes se encarga de mantener el estado deseado, realizando los cambios y ajustes necesarios

4. Escalabilidad: Kubernetes proporciona escalabilidad horizontal de las aplicaciones añadiendo o eliminando automáticamente réplicas de contenedores en función de la carga y la demanda. Además, Kubernetes proporciona mecanismos de recuperación, reiniciando automáticamente los contenedores o los nodos en caso de fallos.

Se obtiene una mayor abstracción, automatización, escalabilidad y recuperación, permitiendo una gestión más eficiente del sistema.

## 5.2. Descripción de un flujo de trabajo

Este apartado tiene como objetivo describir cómo son los **workflows** que se generan para esta aplicación: tareas, dependencias, contenedores, etc.

El punto de partida es la definición de un experimento climático en JSON, como en el siguiente ejemplo:

**Código 1: Ejemplo definición de un experimento en JSON**

```
{
  "id":0,
  "name":"test",
  "steps":[
    {
      "id":"reference",
      "type":"catalog_datasource",
      "dataset":"WFDE5",
      "variables":[
        "pr"
      ],
      "spatialCoverage":"POLYGON ((-4.4476258494631455 42.552732150536855,
          -4.4476258494631455 43.91774384946314, -3.0745481505368546
          43.91774384946314, -3.0745481505368546 42.552732150536855,
          -4.4476258494631455 42.552732150536855))",
      "startDate":"1980-01-01",
      "endDate":"2015-01-01"
    },
    {
      "id":"calibration",
      "type":"catalog_datasource",
      "dataset":"CMIP6",
      "variables":[
        "pr"
      ],
      "spatialCoverage":"POLYGON ((-4.4476258494631455 42.552732150536855,
          -4.4476258494631455 43.91774384946314, -3.0745481505368546
          43.91774384946314, -3.0745481505368546 42.552732150536855,
          -4.4476258494631455 42.552732150536855))",
      "models":[
```

```
        "UKESM1-0-LL_r1i1p1f2"
      ],
      "scenarios":[
        {
          "name":"historical",
          "startDate":"1980-01-01",
          "endDate":"2015-01-01"
        }
      ],
      "mask":"land",
      "startDate":"1980-01-01",
      "endDate":"2015-01-01"
    },
    {
      "id":"projection",
      "type":"catalog_datasource",
      "dataset":"CMIP6",
      "variables":[
        "pr"
      ],
      "spatialCoverage":"POLYGON ((-4.4476258494631455 42.552732150536855,
          -4.4476258494631455 43.91774384946314, -3.0745481505368546
          43.91774384946314, -3.0745481505368546 42.552732150536855,
          -4.4476258494631455 42.552732150536855))",
      "models":[
        "UKESM1-0-LL_r1i1p1f2"
      ],
      "scenarios":[
        {
          "name":"ssp585",
          "startDate":"2015-01-01",
          "endDate":"2019-01-01"
        }
      ],
      "mask":"land"
    },
    {
      "id":"bias_adjustment",
      "type":"bias_adjustment",
      "method":"eqm",
      "validation":"NONE",
      "params":{
        "wet.threshold":1.0,
        "precipitation":true,
        "extrapolation":"none",
        "n.quantiles":50
      },
      "dependsOn":[
        "reference",
        "projection",
        "calibration"
      ]
    },
    {
      "id":"post_processing",
      "type":"post_processing",
      "temporalResolution":"DAILY",
      "temporalResolutionAggregationFunction":{
```

```
      },
      "spatialResolutionAggregation":null,
      "spatialResolutionAggregationFunction":{
      },
      "fileFormat":"NETCDF",
      "dependsOn":[
        "bias_adjustment"
      ]
    }
  ],
  "versions":{
    "pyss":"latest",
    "pyagg":"latest",
    "climate4r":"latest"
  }
}
```

A partir de esta definición, hay que generar la definición de un workflow (también como fichero JSON) para ser enviada al servidor de Argo (Argo Workflows). Esta definición del workflow tiene como metadatos información como el nombre del workflow o el *namespace*. En la especificación ya se define el DAG (*directed acyclic graph*) con sus tareas, dependencias, volúmenes de datos y plantillas de contenedores.

En general, las tareas ejecutadas por un workflow son: (1) descargar un zip con ficheros de parámetros de configuración para las tareas, (2) realizar tareas de recuperación de datos y agregación espacial, (3) realizar tareas de ajuste de sesgos, (4) realizar un postprocesamiento de datos, y, por último, (5) guardar la salida en el sistema de almacenamiento.

La primera tarea del DAG se encarga descargar el fichero zip con los ficheros de parámetros del servicio S3 y descomprimirlo en el volumen de datos del *workflow*. Esto lo realiza un contenedor mediante la herramienta **rclone** [24] (herramienta de línea de comandos para sincronizar y transferir datos entre diferentes sistemas de almacenamiento en la nube y sistemas de archivos locales). La siguiente tarea depende de si se ha especificado un fichero de agregación espacial. En este caso, también se hace uso de un contenedor mediante la herramienta rclone para descargar este fichero adicional.

Los siguientes apartados detallan las tareas necesarias para las fases de recuperación de datos, ajuste de sesgos y postprocesamiento.

#### Tareas de *catalog datasource*/*custom datasource*

Las tres primeras tareas del experimento climático se realizan en paralelo (se corresponden con las etapas *calibration*, *reference* y *projection* indicadas en la figura 1) y tienen como dependencia la tarea anterior de descarga de ficheros de parámetros de configuración (se entiende como dependencia el hecho de que una tarea no puede comenzar si no ha finalizado esta). En el caso de que se haya especificado un fichero de agregación espacial, la dependencia será la tarea de descarga de ese fichero. Estas tareas consisten en la descarga de datos (ficheros NetCDF) del S3. La lista de *chunks* a descargar se ha calculado previamente haciendo una consulta al catálogo y ha quedado registrada en los ficheros de parámetros. Los chunks sirven para dividir grandes cantidades de datos en partes más pequeñas y manejables. Cada chunk representa un conjunto de datos específico que puede ser consultado y analizado de manera independiente. Además, los chunks permiten un acceso más rápido a los datos, ya que se pueden cargar de manera pa-

ralela. La información de cada chunk incluye el nombre del bucket, su localización dentro del bucket, tamaño, etc. Estas tareas las realiza un contenedor cuya imagen es el **Java Downloader**.

En siguiente lugar, se separan los chunks por tipo de tarea de descarga (*projection*, *calibration* y *reference*), por escenario y por modelo. Para cada uno de estos casos, generamos una tarea de selección espacial. Cada una de estas tareas tiene como dependencia la tarea de descarga de datos correspondiente. Estas tareas las realiza el **Python Spatial Selector** (componente responsable de preparar datos para una región dada de los diferentes fragmentos disponibles localmente en el nodo informático y convertirlo en el formato de entrada que necesita el siguiente paso).

#### Tareas de *bias adjustment*

Las siguientes tareas son de *bias adjustment*, una por cada escenario de proyección y modelo. Cada una de estas tareas recibe como parámetro un fichero con la localización de tres ficheros: fichero de referencia (con el modelo correspondiente), fichero de calibración (con el modelo correspondiente y escenario histórico) y fichero de proyección (con el modelo y escenario de proyección correspondiente). Por tanto, cada una de estas tareas tiene como dependencias las tres tareas de selección espacial correspondientes al anterior paso. Estas tareas las realizan contenedores **climate4RWrapper**. Para cada una de estas tareas, se añade una tarea adicional, cuya dependencia es la tarea anterior. Las realiza un contenedor **Python Aggregator**.

Es necesario destacar que si el experimento requiere validación **BASIC** o **ADVANCED**, hay que añadir nuevas tareas. Estas tareas únicamente requieren dos ficheros de datos: fichero de referencia y fichero de calibración (el escenario es histórico). Las realiza igualmente **climate4RWrapper**, y también es necesario realizar la fusión de los datos fragmentados en un solo archivo (realizado de la misma manera por **Python Aggregator**). Estas tareas tienen como objetivo determinar si los modelos utilizados para el ajuste son capaces de reproducir correctamente los patrones climáticos observados en los datos históricos.

#### Tareas de *post processing*

La siguiente tarea es producir el formato de salida final. Para realizar esta tarea es necesario que hayan acabado las tareas descritas en el anterior apartado. Esta tarea también la realiza el **Python Aggregator**. Este componente también realizará varias operaciones de reducción de dimensiones, que dependerá de las demandas del usuario. Estas operaciones pueden incluir agregaciones espaciales en áreas específicas (polígonos), así como reducción de resolución temporal. Si se hace validación se añade una tarea similar que tiene como dependencias las tareas de validación del apartado anterior.

Por último, la última tarea consiste en subir los resultados finales del *workflow* al S3 (disponibles en el volumen de datos). La tarea es realizada por un contenedor **rclone**. En el *entrypoint* del contenedor especificamos qué ficheros del volumen de datos del *workflow* se suben al S3.

En la figura 4 se ve el esquema de tareas de un DAG. En este, se muestra la tarea *download-zip* (descarga de ficheros de parámetros del S3), después aparecen otras 6 tareas (tareas de descarga), continúa con las dos tareas de BA, y acaba con una tarea de postprocesado y la subida de los resultados al S3.


Figura 4: Grafo de tareas de un flujo de trabajo

## 5.3. Almacenamiento de datos

Para compartir datos entre las diferentes tareas de los *workflows* y almacenar la base de datos del catálogo se utilizan volúmenes Kubernetes. Se necesitarán tres tipos de volúmenes: volumen persistente, volumen efímero y volumen a partir de ConfigMap.

El volumen persistente se utiliza para almacenar la base de datos del catálogo. Se crea un PVC (*Persistent Volume Claim*) que solicite un cierto tipo de almacenamiento, y Kubernetes lo asigna automáticamente a un PV (*Persistent Volume*) que cumpla con los requisitos del PVC. Se crea en el clúster de Kubernetes, no lo hace el *runner*.

Un volumen efímero es un tipo de volumen que se crea y se destruye junto con el *workflow* en el que se define. Los volúmenes efímeros se utilizan para almacenar datos temporalmente, y se eliminan automáticamente cuando el *workflow* finaliza. Utilizaremos un volumen efímero para almacenar tanto los ficheros de parámetros como los ficheros NetCDF intermedios de las diferentes tareas. Este volumen se define en la especificación del Workflow (sección `volumeClaimTemplates`). El tamaño que tendrá este volumen dependerá del tamaño del experimento. Este tamaño se estima a partir del tamaño de los *chunks* que se obtienen en los pasos de tipo *catalog datasource*. Los *chunks* que existen en la base de datos del catálogo tienen un atributo "size", que indica el tamaño en bytes del *chunk*. Con esta información, y sabiendo que existen tres fases en un *workflow* (*catalog datasource*, *bias adjustment* y *post processing*) y que en cada una se generan

ficheros intermedios, especificaremos el siguiente tamaño de volumen (dejando cierto margen):

$$size = (chunksCalibration + chunksProjection + chunksReference) * 4 \quad (1)$$

Por otro lado, en el volumen se especifican los modos de acceso. Los modos de acceso son propiedades que se utilizan para especificar cómo se puede acceder al volumen. Utilizaremos el modo de acceso **ReadWriteMany** (RWX). Este modo de acceso permite que el volumen se pueda montar como de lectura y escritura en múltiples pods de Kubernetes al mismo tiempo. De forma nativa, Kubernetes solo soporta los modos de acceso **ReadWriteOnce** y **ReadOnlyMany**. Esto significa que un volumen solo puede ser montado en un pod a la vez o en múltiples pods en modo de solo lectura. Si se necesita un volumen que pueda ser montado en múltiples nodos y también pueda ser modificado, entonces se necesita una solución de almacenamiento que ofrezca soporte para acceso ReadWriteMany.

**Longhorn** es una solución de almacenamiento de bloques distribuida y de código abierto para Kubernetes que proporciona funcionalidades adicionales para manejar volúmenes con este modo de acceso. Proporciona una solución resistente a fallos y está diseñada para proteger los datos de aplicaciones críticas. Longhorn se ejecuta como un controlador de almacenamiento nativo de Kubernetes y se integra fácilmente con la infraestructura existente. Además, Longhorn también ofrece funciones como instantáneas, clonación y replicación para ayudar en la gestión de datos.

El otro volumen que se utilizará será un ConfigMap. Un ConfigMap se puede utilizar como un volumen en un pod de Kubernetes. Cuando se monta un ConfigMap como un volumen en un pod, los datos se exponen como archivos en el sistema de archivos del contenedor. Esto permite que los contenedores tengan acceso a la configuración sin tener que cambiar su imagen o código fuente. Es necesario para los contenedores que sincronizan archivos y directorios entre diferentes proveedores de almacenamiento en la nube. El ConfigMap contiene información de configuración sobre el sistema de almacenamiento en la nube Amazon S3.

Utilizamos Amazon S3 para almacenar la definición del *workflow* en formato JSON, los ficheros de parámetros, la salida final del *workflow* y la traza de las tareas. Dentro del S3, existe un bucket específico para guardar esta información.

# 6. Implementación

## 6.1. Configuración del entorno de desarrollo

La fase de desarrollo se realiza en un clúster local de Kubernetes de solamente un nodo. Un nodo es una máquina de trabajo en Kubernetes, y puede ser una máquina virtual o física. En este caso, se trata de una máquina física, el equipo local. Kubernetes, pese a que permite desplegar aplicaciones en contenedores aislados, tiene una puesta en marcha compleja. Para simplificar este proceso, surgió **K3s**. K3s es una distribución de Kubernetes mucho más liviana y con menor consumo de recursos. Para ello, se ha prescindido de componentes del núcleo de Kubernetes, que pueden ser recuperados a través de extensiones. Una explicación más detallada sobre el proceso de instalación se puede encontrar en [22].

Otro componente del entorno es **kubectl**, que es una herramienta de línea de comandos que se utiliza para interactuar con clústeres de Kubernetes.

Para que kubectl encuentre y acceda a un clúster de Kubernetes, necesita un archivo de configuración, que se crea automáticamente cuando creas un clúster. De forma predeterminada, la configuración de kubectl se encuentra en el fichero `~/.kube/config`. Para verificar que kubectl está configurado correctamente, se puede obtener el estado del clúster con el comando `kubectl cluster-info`.

```
cuem@santabot:~$ kubectl cluster-info
Kubernetes control plane is running at https://127.0.0.1:6443
CoreDNS is running at https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/kube-dns:
dns/proxy
Metrics-server is running at https://127.0.0.1:6443/api/v1/namespaces/kube-system/services/ht
tps:metrics-server:https/proxy
```

Figura 5: Verificación de la configuración de kubectl

En el clúster local se instala **Argo Workflows** (más detalles de la instalación en https://argoproj.github.io/argo-workflows/quick-start/). Se instala un controlador y un servidor (incluye una interfaz de usuario basada en la web que proporciona una vista gráfica de las aplicaciones implementadas en el clúster de Kubernetes). En la instalación, tanto el controlador como el servidor se despliegan en el namespace `argo`. El servidor de Argo utiliza de forma predeterminada la autenticación de cliente, que requiere la autenticación de usuarios. Durante el desarrollo, se cambia el modo de autenticación de servidor para que podamos pasar por alto el inicio de sesión de interfaz de usuario. También es importante asegurarse de que la opción base de instalación es *cluster install*, que hace que el servidor atienda y ejecute *workflows* en todos los *namespaces*. Para acceder al servidor desde otros sitios se crea un servicio Kubernetes en el namespace **argo**.

**Código 2: YAML del servicio Argo**

```yaml
apiVersion: v1
kind: Service
metadata:
  name: argo-service
  namespace: argo
spec:
  type: LoadBalancer
  selector:
```

```
    app: argo-server
  ports:
    - protocol: TCP
      port: 8080
      targetPort: 2746
```

A través de este servicio se puede acceder a la UI de Argo en `https://localhost:8080/`.

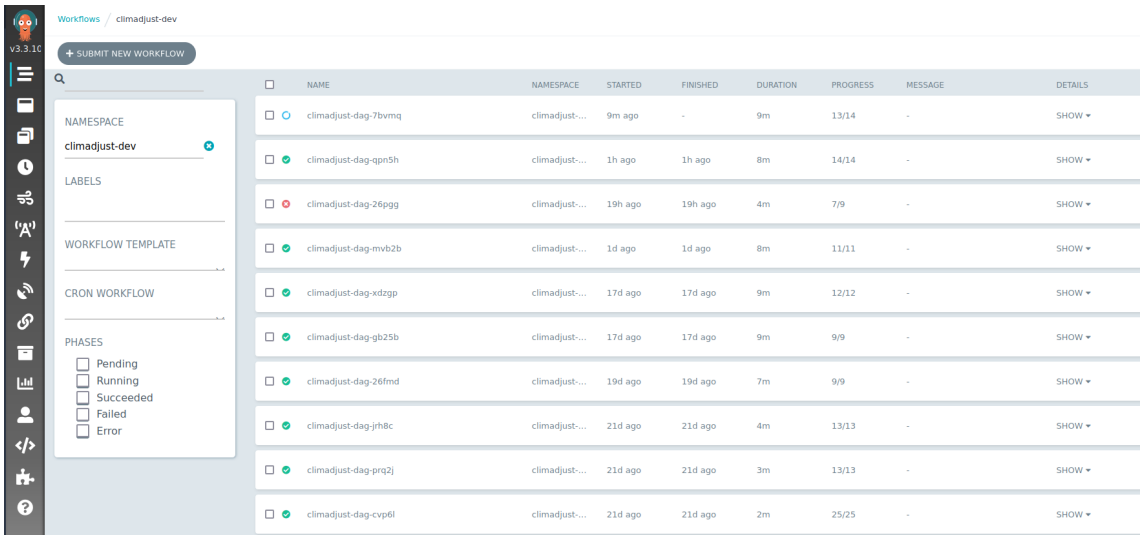


Figura 6: Interfaz de usuario de Argo

Es común utilizar diferentes *namespaces* para crear diferentes entornos aislados. Durante el desarrollo se trabaja con dos *namespaces*: **argo** (controlador y servidor Argo) y **climadjust-dev** (objetos Kubernetes de nuestra aplicación).

Es importante tener instalado **Longhorn** en el nodo con kubectl. Longhorn [23] es un sistema de almacenamiento de bloques distribuido que extiende las funcionalidades de Kubernetes para manejar volúmenes con otros modos de acceso que son útiles para este proyecto.

La configuración de los repositorios de código donde se están los componentes desarrollados (denominados **Argo Runner** y **Jdl**). Implica la creación de un archivo `.gitlab-ci.yml`, que es la configuración de GitLab CI/CD para automatizar la integración y entrega continua de un proyecto. Contiene una lista de tareas llamadas "job" que se ejecutan en un pipeline definido. Un pipeline es una serie de etapas y trabajos que se ejecutan en una secuencia definida. En él se especifican los scripts y comandos que se ejecutan en cada trabajo.

También existe un fichero `settings.xml` en cada proyecto, y en la sección `<servers>` se configuran los servidores utilizados por Maven para descargar y cargar artefactos, así como para autenticarse y autorizarse en esos servidores. Existe un repositorio que incluye librerías creadas por la empresa que son necesarias para este proyecto, así como un registro de imágenes para almacenar las imágenes de las aplicaciones.

**Código 3: Fichero XML creedenciales del repositorio de librerías**

```
<settings>
    <servers>
        <server>
```

```
            <id>docker.nexus.predictia.es</id>
            <username>$${env.NEXUS_USER}</username>
            <password>$${env.NEXUS_PW}</password>
        </server>
        <server>
            <id>predictia-releases-repo</id>
            <username>$${env.NEXUS_USER}</username>
            <password>$${env.NEXUS_PW}</password>
        </server>
    </servers>
</settings>
```

## 6.2. Implementación del nuevo motor de cómputo

Este apartado se centra en describir la implementación del principal componente desarrollado en este proyecto, denominado Argo Runner. Se trata del motor de cómputo encargado de generar diferentes tareas a partir de la definición de un experimento y exponer una API para consultar el estado de las mismas.

### 6.2.1. Estructura de directorios

Los ficheros del proyecto Java están organizados en paquetes. Los paquetes sirven para agrupar clases relacionadas y definen un espacio de nombres para las clases que contienen. Se sigue la siguiente estructura de directorios (únicamente se muestran algunas de clases utilizadas en el proyecto):

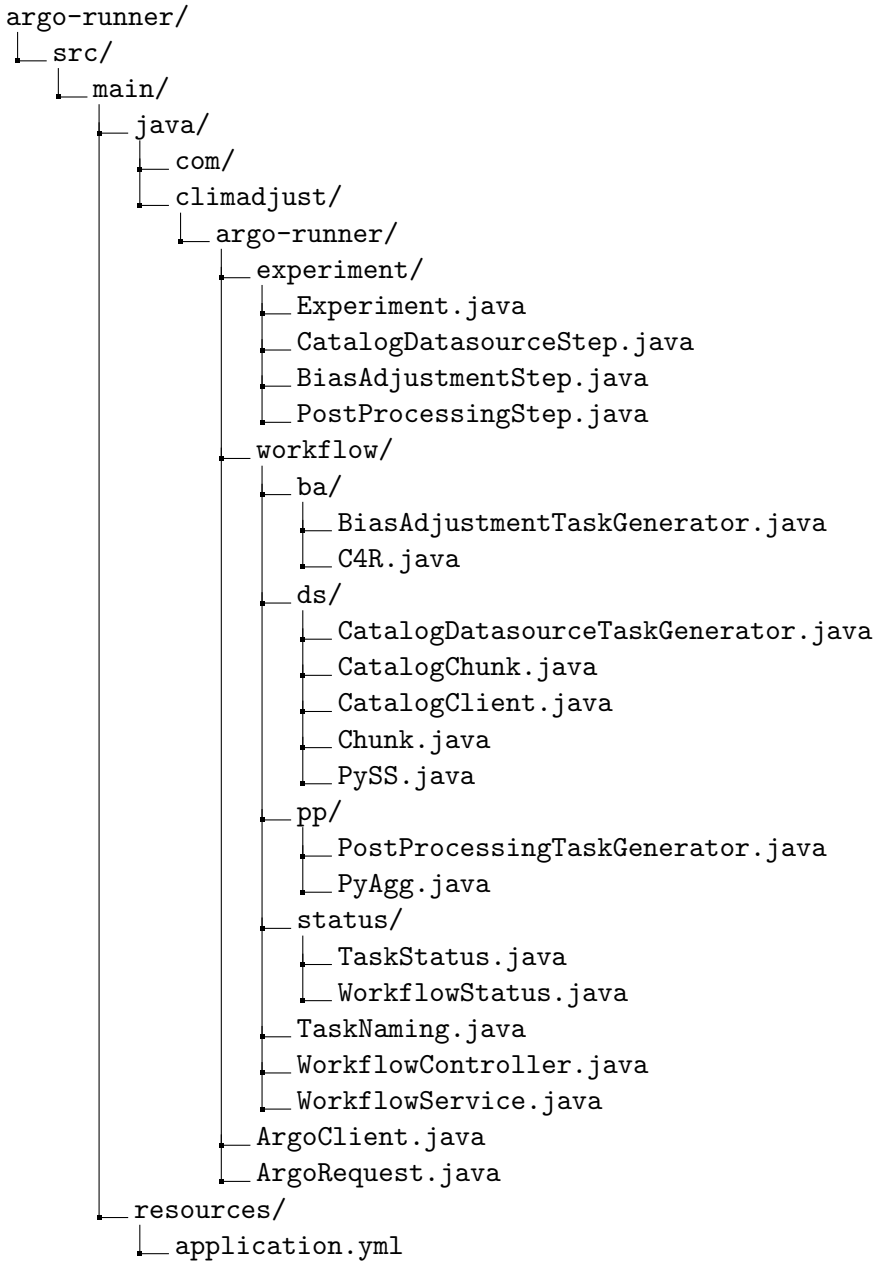


Figura 7: Estructura de directorios del componente Argo Runner

En el directorio **src/main/java** se ubica el código fuente de la aplicación y en **src/main/resources** se incluyen recursos como archivos de configuración o archivos de propiedades.

El paquete **experiment** incluye clases necesarias para hacer el mapeo del JSON del experimento. También existen otras dos clases, la clase **ArgoClient.java** contiene métodos para interactuar con el servidor de Argo, mientras que la clase **ArgoRequest** es una clase para crear y enviar solicitudes a Argo. A través de esta clase es posible enviar solicitudes para crear flujos de trabajo.

Por otra parte, el paquete **workflow** contiene todas las utilidades relacionadas con los *workflows*. La clase **WorkflowController.java** es un controlador web encargado de procesar solicitudes

HTTP de creación, eliminación o consulta de la ejecución de flujos de trabajo. Además, existe una clase **WorkflowService.java** con funcionalidades para interactuar con el S3, generar el cuerpo de la petición HTTP para crear *workflows* o esperar a que acabe el *workflow* para guardar la traza. Se ha añadido una clase denominada **TaskNaming.java** que implementa la lógica para establecer el nombre de las tareas (se establece según el tipo de tarea, variable, modelo, escenario, etc.). Este paquete contiene otros subpaquetes. Estos subpaquetes contienen clases para generar tareas o plantillas de contenedores, así como para definir clases que ayuden a la generación de ficheros de parámetros para las diferentes tareas. El subpaquete **ds**, además, incluye clases que ayudan a interactuar con el catálogo y gestionar el cuerpo de las peticiones que se hacen al mismo.

### 6.2.2. Gestión de configuración a través del archivo application.yml

El fichero application.yml es el archivo de Spring Boot que se utiliza para establecer las propiedades de la aplicación.

**Código 4: Fichero de configuración de Argo Runner**

```
s3:
  poolSize: 4
  endpoint:
  region:
  key:
    id:
    secret:
  secret:
    name: s3-config
    key: defaultKey
    secret: defaultSecret
  fallback:
    secret:
      name: s3-config
      key: fallbackKey
      secret: fallbackSecret
argo:
  host: argo-service.argo
  port: 8080
  uri: "https://${argo.host}:${argo.port}"
docker:
  registry: docker.nexus.predictia.es
server:
  port : 8081
catalog:
  host: catalog-web
  port: 28080
  url: "http://${catalog.host}:${catalog.port}"
workflow:
  namespace: climadjust-dev
  generateName: climadjust-dag-
  bucket:
    name: "bas4cds-experiments-test"
  paramsData:
    path: ${java.io.tmpdir}/experiments
  spool:
    path: /workflow
    volumeName: spool
    volumeStoreClass: longhorn
  jdlContainer:
    image: ${docker.registry}/bas4cds/jdl
    command: ["java"]
```

```
      args: ["-jar", "/app.jar", "--input_file={{inputs.parameters.input_file}}",
          "--output_folder={{inputs.parameters.output_folder}}"]
      resources:
        requests:
          memory: 200Mi
          cpu: 100m
        limits:
          memory: 500Mi
          cpu: 100m
  ...
  gc:
    volume: OnWorkflowSuccess
    pod: OnWorkflowSuccess
    artifact: OnWorkflowDeletion
```

Este fichero de configuración contiene información sobre la configuración del servicio de almacenamiento en la nube (puede haber valores vacíos que se especifican posteriormente con variables de entorno), el acceso al servicio de argo, el registro de imágenes Docker, el acceso a la API del catálogo y gestión del recolector de basura para los recursos de Argo.

Por otro lado, tenemos configuración del *workflow*. Esto incluye información como el nombre del bucket del S3, ruta de los ficheros de datos, gestión de volúmenes y configuración de los contenedores: nombre de la imagen, comandos, argumentos y recursos de CPU y memoria.

### 6.2.3. API para gestionar la creación y estado de los flujos de trabajo

Para poder lanzar y controlar el estado de los flujos de trabajo, la aplicación expone una API para poder monitorizarlos desde componentes externos. Las opciones implementadas para los workflows son: creación, consulta del estado de ejecución, reintento de tareas fallidas, eliminación y consulta del estado de las tareas.

La API se implementa con un controlador Spring Boot. Es una clase que se encarga de recibir las solicitudes HTTP, procesarlas y devolver una respuesta al cliente. La anotación `@Controller` en una clase indica que esta se va a utilizar como controlador de Spring. Esta anotación se utiliza en conjunto con otras anotaciones como `@RequestMapping`, `@ResponseBody` o `@GetMapping`.

#### Creación de un nuevo workflow

Cuando llega una petición para realizar un nuevo experimento, el sistema hace lo descrito en el esquema de la figura 8.

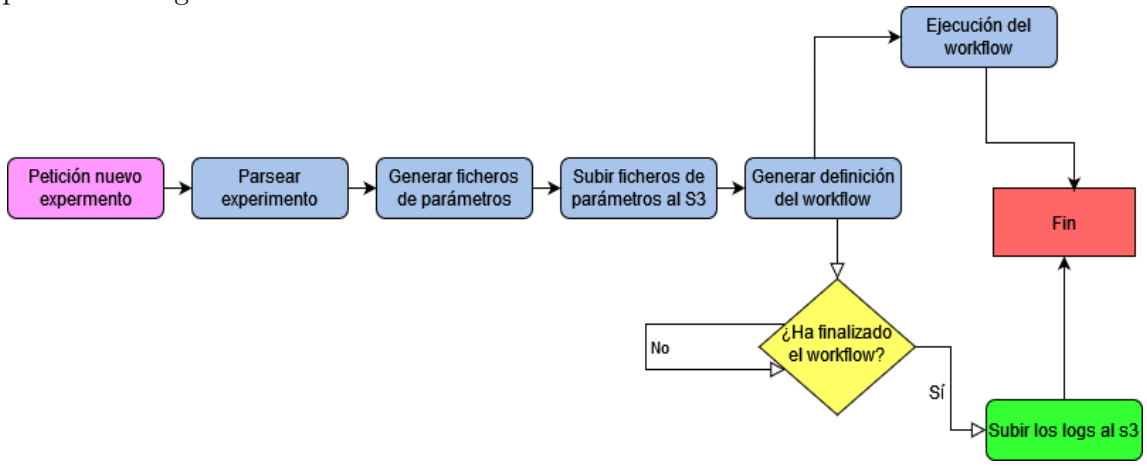


Figura 8: Diagrama de flujo proceso de petición de realización de un experimento

La creación de un nuevo workflow se realiza a través de una petición POST. En el cuerpo de la petición se especifica la definición del experimento. Además, requiere como parámetro la prioridad con la que se va a lanzar el experimento. La prioridad puede ser **STANDARD** o **HIGH** y se define a nivel de workflow. En la especificación del workflow se añade el atributo *priority*, que indica la prioridad relativa con la que se debe programar su ejecución en el clúster. El valor de este atributo es un número entero, y se establece un mayor valor para prioridad **HIGH** que para **STANDARD**. También se establece la prioridad a nivel de pod. Con la característica *podPriorityClassName* asignamos los pods de un workflow a una clase de prioridad definida en el clúster. Esto hace que los pods de los workflows con mayor prioridad se programen antes que los pods con menor prioridad. La clase de prioridad definida para prioridad alta es la siguiente:

**Código 5: Clase de prioridad para workflow con prioridad alta**

```
apiVersion: scheduling.k8s.io/v1
description: Contains high priority climadjust jobs
kind: PriorityClass
metadata:
  name: climadjust-high-pc
preemptionPolicy: Never
value: 100
```

Con la definición del experimento y su prioridad, hace el mapeo a un objeto **Experiment**, y con este, genera los ficheros de parámetros, los sube al S3, genera la especificación de un DAG incluyendo las tareas, plantillas, volúmenes, etc. necesarios, y por último, envía esta especificación a Argo para la ejecución del workflow. Las tareas se definen en base a lo descrito en apartados anteriores. La respuesta que retorna es la URL del workflow, con la que se puede hacer una consulta de su estado de ejecución. Para poder atender varias peticiones a la vez sin bloquear el servicio, se utiliza la interfaz **ExecutorService**. Se trata de una interfaz en Java que proporciona un framework para ejecutar tareas en segundo plano de forma asíncrona. Permite la creación de hilos para ejecutar tareas en paralelo y gestionarlos de forma eficiente.

Uno de los hilos que se utiliza es para esperar a la finalización del workflow para guardar los logs de las tareas en el almacén de objetos en la nube. Si ha finalizado, sube los logs generados por las tareas al S3, en la localización escogida para guardar la información de ese workflow en concreto. Los logs son registros que generan las aplicaciones mientras se ejecutan. Contienen información sobre errores, excepciones, advertencias o eventos que ocurren en la aplicación. Son importantes sobre todo para comprender lo que sucede en la aplicación durante la ejecución e identificar errores que se puedan producir.

Para poder obtener los logs hay que configurar `archiveLogs` en el workflow-controller configmap en el namespace Argo, en la especificación del workflow o a nivel de plantilla. En este proyecto configuramos los logs en la especificación del workflow. También hay que configurar un repositorio de artefactos para indicar dónde se guardan. Los artefactos son archivos utilizados por los flujos de trabajo. Se pueden definir artefactos como entradas o salidas del flujo de trabajo. Las entradas de artefactos se utilizan para proporcionar datos al flujo de trabajo, mientras que las salidas de artefactos se utilizan para almacenar los resultados de tareas, en este caso los logs. Cuando finaliza un pod, de forma automática se guardan sus logs en el repositorio de artefactos configurado.

### Consulta del estado de un workflow

Otra de las funcionalidades que ofrece la API es consultar el estado de ejecución de un workflow. Argo guarda un histórico de todos los workflows de todos los namespaces. La consulta se realiza a través de una petición GET en la que se incluye el nombre del workflow. Argo retorna un JSON con información muy detallada, incluyendo la propia especificación o los pods que han acabado. Sin embargo, la aplicación tiene que retornar un objeto que contenga únicamente la siguiente información: identificador, número total de tareas, número de tareas completadas, cantidad de memoria procesada en Gigabytes, estado, prioridad, uso de CPU, localización de los resultados, fecha y hora de comienzo, y fecha y hora de finalización.

El identificador es el nombre que genera Argo (es único en cada caso), el número de tareas es el total de tareas del DAG y las tareas completadas son las tareas cuyo pod encargado de realizarla está en estado "finished".

Por otro lado, es importante obtener de forma adecuada los recursos consumidos durante la ejecución. Argo proporciona una estimación de cuántos recursos ha consumido cada tipo. Se calcula en base a la duración del contenedor y los límites de recursos establecidos para ese contenedor. Cada indicador se divide por un denominador común según el tipo de recurso. Esta cantidad está expresada en duración en segundos de ese recurso. Utilizamos esta estimación para calcular la cantidad de CPU consumida. Para la cantidad de memoria consumida, el valor que vamos a devolver es el tamaño en GB del volumen efímero (el volumen para compartir datos entre las tareas).

### Reintentar tareas fallidas

Durante la ejecución de un flujo de trabajo se pueden producir errores. Un *retry* (reintento) supone volver a ejecutar las tareas que han fallado. Es de una forma de reintentar las tareas que han fallado, evitando la necesidad de crear un nuevo flujo de trabajo desde cero. Este reintento se va a realizar de forma manual, únicamente se realizará si así lo requieren los componentes que hacen uso de la API.

### Eliminar un workflow

Eliminar un workflow supone detener su ejecución y eliminar todos los recursos asociados al mismo, incluyendo los pods que se están ejecutando. En este caso se eliminan los pods, el volumen de datos que se está utilizando y los logs. Una vez eliminado un workflow, no se puede recuperar su estado ni su información asociada. Además, también hay que eliminar la información guardada en el S3 relacionada con el workflow. Esto es, los ficheros de parámetros, la definición del workflow en JSON y los resultados finales.

Aparte de poder eliminar workflows en ejecución, también se pueden eliminar workflows que han finalizado correctamente y los que han finalizado con errores. En ese caso, se realiza el mismo proceso pero sin tener que eliminar ningún pod, porque en ese momento no hay proceso en curso.

### Consultar el estado de las tareas de un workflow

La API implementa la funcionalidad de poder consultar qué tareas han finalizado y cuáles quedan por finalizar. Esta capacidad es útil para que el usuario pueda ver en qué estado se encuentra la ejecución. Por cada tarea del DAG, obtenemos el nombre de la tarea, su estado, las dependencias con otras tareas, cuándo empezó, cuándo ha acabado (en caso de que haya finalizado) y el tipo de paso del experimento al que pertenece la tarea.

Casi toda esta información se puede obtener a partir la información que guarda Argo de cada flujo de trabajo. Sin embargo, para saber el tipo de paso del experimento al que corresponde cada tarea, se ha implementado una lógica que lo infiere a partir del nombre de la tarea.

### 6.2.4. Uso de APIs HTTP mediante clientes Java

Un cliente HTTP es una aplicación que envía solicitudes HTTP a un servidor web y recibe las respuestas correspondientes. Se utiliza un cliente HTTP para hacer peticiones al servidor de Argo, y así poder lanzar un *workflow*, consultar su estado, el estado de las tareas o eliminar workflows.

Para comunicarse con otros servicios desde el *runner* se utilizan APIs HTTP. Para ello se necesita un cliente HTTP. OkHttp es una biblioteca de cliente HTTP de código abierto para Java. Las peticiones se realizan a través de un objeto denominado **Request**. Necesita, al menos, una URL de destino y un método HTTP (como GET, POST, PUT, DELETE, etc.). Además, puede incluir información adicional, como encabezados de solicitud, parámetros de consulta, cuerpo de solicitud y otros datos según sea necesario. El método especificado en el fragmento de código 6 sirve para hacer una petición de ejecución de *workflow* al servidor de Argo.

**Código 6: Método para la petición de ejecución de un workflow**

```
public String createWorkflow(ArgoRequest request) throws Exception {
    var createUri = URI.create("%s/api/v1/workflows/%s".formatted(argoApiServer,
        namespace));
    var json = OBJECT_MAPPER.writeValueAsString(request);
    log.info("Submitting to {}: {}", createUri, json);
    var httpRequest = new okhttp3.Request.Builder()
            .url(createUri.toURL())
            .header("Content-Type", "application/json")
            .header("Accept", "application/json")
            .post(RequestBody.create(json, MediaType.get("application/json;
                charset=utf-8")))
            .build();
    try(var response = client.newCall(httpRequest).execute()){
        log.info("Server response {}", response.code());
        var responseTree = OBJECT_MAPPER.readTree(response.body().string());
        if(response.code() >= 400){
            log.warn("Error creating workflow: {}", responseTree.get("message"));
            throw new Exception("Error creating workflow");
        }else{
            var name = responseTree.get("metadata").get("name").asText();
            log.info("Created workflow with name {}", name);
            return name;
        }
    }
}
```

### 6.2.5. Procesamiento de JSON y YAML con Jackson en Java

En la implementación se ha visto la necesidad de trabajar con ficheros en formato JSON o YAML. JSON se utiliza para generar el cuerpo de las peticiones de creación de flujos de trabajo, y para obtener la definición de un experimento. Por otro lado, YAML se utiliza para generar los ficheros de parámetros de las tareas del DAG. Por tanto, es importante trabajar de forma

eficiente con estos formatos de datos.

Se utiliza la funcionalidad de clases **Record** con la librería **Jackson**[21] para Java. Las clases Record fueron introducidas en Java 14, y permiten definir clases de manera compacta. Son una manera abreviada de definir clases que contienen atributos inmutables. La clase incluye métodos de acceso generados automáticamente para los campos, y también una implementación de los métodos *equals*, *hashCode* y *toString*. Por otro lado, Jackson es una librería que se utiliza para leer y escribir datos JSON o YAML en objetos Java y viceversa. Con Jackson, se puede hacer uso de la clase `ObjectMapper`. Se pueden configurar varias opciones de serialización y deserialización utilizando ObjectMapper, como el formato de fecha, el manejo de nulos o la inclusión de campos.

### 6.2.6. Medidas de seguridad para gestión de datos confidenciales

Es un requisito indispensable mantener la seguridad de los datos confidenciales. Para gestionar estos datos sensibles, Kubernetes proporciona dos recursos: secrets y configmaps. En el clúster Kubernetes se crea un secret que contiene las claves para acceder al S3. También se crea un configmap que contiene información general no confidencial sobre el S3 (por ejemplo, la localización y el endpoint). Los secrets se utilizarán en los contenedores como variables de entorno, mientras que los configmaps se utilizarán en los contenedores a través de un volumen. Se monta el volumen en un directorio dentro del contenedor.

El contenedor **Argo Runner** interactúa con el S3, por lo que necesita que se pasen los secets definidos en el clúster como variables de entorno. Además, los contenedores **rclone** necesitan los secretos del S3. Rclone almacena las credenciales en el archivo de configuración ubicado en */config/rclone/rclone.conf*, por lo que se monta el configmap en un volumen y se pasan los secretos como variables de entorno. De esta manera, toda la información necesaria estará disponible en el fichero *rclone.conf* del contenedor.

### 6.2.7. Recursos utilizados por las tareas y gestión del recolector de basura

Uno de los requisitos del sistema es que a cada contenedor encargado de realizar una tarea se le tienen que asignar unos límites de recursos. Los límites de recursos controlan cuánta CPU y memoria puede usar cada contenedor. Para especificar los límites de recursos en Argo, se puede utilizar la sección *resources* dentro de la definición de un contenedor. Los límites que se establecen afectan al rendimiento de los contenedores, y pueden causar que un contenedor se bloquee o se reinicie si se queda sin recursos. Los límites que se han establecido para los contenedores se adecúan a la carga de trabajo prevista para cada uno de ellos. Por ejemplo, los contenedores que utilizan rclone transfieren un número pequeño de ficheros, por lo que no necesitan mucha memoria ni mucha CPU. Sin embargo, el contenedor **climate4R** realiza ajuste de sesgos y para ello necesita una cantidad elevada de memoria. La tabla muestra las plantillas utilizadas (junto con la imagen del contenedor y los límites de recursos establecidos).

Tabla 4: Plantillas utilizadas en los flujos de trabajo

| Nombre | Imagen | Memoria | CPU |
|---|---|---|---|
| jdlContainer | jdl | 500MB | 100mi |
| pySSContainer | pyss | 2Gi | 100mi |
| pyAggContainer | pyagg | 2Gi | 100mi |
| downloadZipContainer | rclone/rclone:1.61 | 200Mi | 100mi |

| | | | |
|---|---|---|---|
| c4rContainer | climate4r | 5Gi | 1000mi |
| uploadOuputContainer | rclone/rclone:1.61 | 2Gi | 200mi |
| downloadFileContainer | rclone/rclone:1.61 | 2Gi | 200mi |

Otro aspecto a tener en cuenta es la eficiencia. Es necesario no sobrecargar al clúster con pods y volúmenes que ya no se utilizan. Para ello, en la especificación del workflow se indica la gestión del recolector de basura. Al finalizar la ejecución del workflow, se eliminan los pods y los volúmenes efímeros creados. Además, si se elimina el workflow, también hay que eliminar los artefactos generados.

## 6.3. Implementación del componente de descarga de datos

Este subsistema, denominado **Java Downloader**, tiene como finalidad descargar los datos necesarios para realizar los experimentos. En la arquitectura del sistema antes de realizar la migración a Kubernetes no existía este componente, estaba integrado dentro del JRM. Sin embargo, se ha considerado necesario separar esta funcionalidad en un componente independiente. De esta manera, realiza las tareas representadas con el nombre *Jdl* en el componente *computing engine* de la figura 3. Se encarga de descargar los ficheros NetCDF necesarios para hacer el ajuste de sesgos. Los datos se descargan del S3. Recibe como parámetros un fichero de configuración (contiene información de los chunks que tiene que descargar), y la ruta donde guardar los ficheros descargados. El fichero de configuración es un fichero YAML. Se muestra un ejemplo a continuación:

**Código 7: Ejemplo fichero de entrada del Java Downloader**

```
- experiments:
  - "historical"
  members:
  - "UKESM1-0-LL_r1i1p1f2"
  id: 3322590
  size: 2361862
  bucket: "bucket-name"
  location: "CMIP6/UKESM1-0-LL_r1i1p1f2/historical/pr/3322590.nc"
  project: "CMIP6"
  state: "UPLOADED"
  spatialCoverage: "POLYGON ((-30 10, -30 60, 45 60, 45 10, -30 10))"
  variables:
  - "pr"
  timeCoverage:
    start: "1985-01-01"
    end: "1986-01-01"
- experiments:
  - "historical"
  members:
  - "UKESM1-0-LL_r1i1p1f2"
  id: 3322702
  size: 2361862
  bucket: "bucket-name"
  location: "CMIP6/UKESM1-0-LL_r1i1p1f2/historical/pr/3322702.nc"
  project: "CMIP6"
  state: "UPLOADED"
  spatialCoverage: "POLYGON ((-30 10, -30 60, 45 60, 45 10, -30 10))"
  variables:
  - "pr"
  timeCoverage:
```

```
    start: "1991-01-01"
    end: "1992-01-01"
```

El fichero de configuración indica los chunks que hay que descargar. Esta aplicación, por cada chunk, comprueba si está en la memoria caché del S3. Si está en la caché, se obtiene directamente a partir de esta. En otro caso, tiene que ser restaurado. Para ello, a través de un cliente S3 (servicio que interactúa con el proveedor en la nube), se envía una solicitud de restauración.

El proveedor Amazon S3 dispone de un servicio de archivo de datos especialmente diseñado para aquellos datos que no se necesitan acceder constantemente, denominado **Amazon Glacier**. Algunos de los chunks están almacenados en este servicio, y aunque se almacenan sus datos a un coste muy bajo, no se pueden recuperar sus datos inmediatamente cuando se desee. Los datos almacenados en este servicio pueden tardan varias horas en recuperarse. Si se ha esperado un tiempo previamente predefinido y no se ha conseguido recuperar los chunks, se realiza una descarga de los chunks en otro proveedor en la nube alternativo. Por último, se guardan los chunks en la memoria caché que no estuviesen anteriormente.

Por último, se empaqueta la aplicación en una imagen Docker, para poder ser utilizada en tareas de recuperación de datos en el workflow. Se hace a través del siguiente fichero Dockerfile.

**Código 8: Fichero Dockerfile para la generación de imagen Docker**

```
FROM openjdk:17
COPY target/download-runner-0.0.1-SNAPSHOT.jar /app.jar
ENTRYPOINT ["java", "-jar", "/app.jar", "--input_file=params.yml", \
"--output_folder=/data"]
```

## 6.4. Calidad de software: análisis y buenas prácticas

La calidad y el diseño de software son aspectos clave en el desarrollo de aplicaciones informáticas. El análisis y las buenas prácticas son esenciales para garantizar la calidad del software y el buen diseño. El análisis de software implica la evaluación sistemática de un sistema para encontrar problemas, mejorar su calidad y garantizar que cumpla con sus requisitos. Las buenas prácticas son métodos y técnicas recomendadas para mejorar la calidad y el diseño del software. Este apartado cubre aspectos relacionados el análisis del software, buenas prácticas de programación y la refactorización de código.

El diseño de software es una parte crítica para la calidad del producto final. Entre las técnicas empleadas destaca el diseño modular. Consiste en dividir el sistema en componentes independientes. En este caso, se ha optado por separar la funcionalidad de descarga de datos en un componente independiente (imagen Docker). Además, en el *runner* se ha separado la parte de interacción con Argo (a través de un cliente) de la lógica encargada de generar las tareas y plantillas, y separando también la lógica de nombrado de las tareas o de hacer el mapeo del JSON de los experimentos. Otra buena práctica es utilizar patrones de diseño. Uno de los patrones utilizado se denomina ***Coding to interfaces, not implementation***. Consiste en escribir clases basadas en interfaces; la interfaz define el comportamiento del objeto con sus métodos, y después se crea la clase que implementa esos métodos. La interfaz **TaskGenerator** define métodos para generar parámetros, plantillas y tareas.

**Código 9: Interfaz para generar tareas**

```
public interface TaskGenerator<T extends ExperimentStep> {
    Class<T> type();
    default Collection<Path> parameters(Experiment experiment, ExperimentStep
        step) throws IOException{
        if(type().isInstance(step)){
            return generateParameters(experiment, type().cast(step));
        }
        return Collections.emptyList();
    }
    Collection<Path> generateParameters(Experiment experiment, T step) throws
        IOException;
    static Path paramsFolder(WorkflowProps props, Experiment experiment){
        return Paths.get(props.paramsData().path() + "/" + experiment.id());
    }
    static Path spoolFolder(WorkflowProps props, Experiment experiment){
        return Paths.get(props.spool().path() + "/" + experiment.id());
    }
    default Collection<Template> templates(Experiment experiment, ExperimentStep
        step){
        if(type().isInstance(step)){
            return generateTemplates(experiment, type().cast(step));
        }
        return Collections.emptyList();
    }
    Collection<Template> generateTemplates(Experiment experiment, T step);
    default Collection<Task> tasks(Experiment experiment, ExperimentStep step,
        Collection<String> previousStepTasks){
        if(type().isInstance(step)){
            return generateTasks(experiment, type().cast(step), previousStepTasks);
        }
        return Collections.emptyList();
    }
    Collection<Task> generateTasks(Experiment experiment, T step,
        Collection<String> previousStepTasks);
}
```

Otro patrón utilizado es el denominado ***Enumeration Pattern***. Se utiliza para definir un conjunto de valores constantes que un objeto puede tomar. Los *enums* en Java son una implementación de este patrón. Se definen enumerados para describir la prioridad, el tipo de validación o el tipo de agregación espacial.

**Código 10: Ejemplo implementación patrón *Enumeration Pattern***

```
public enum Validation {
    NONE,
    BASIC,
    ADVANCED
}
```

Para analizar la calidad del código se utiliza la herramienta de análisis estático de código **SonarQube** [25]. Esta herramienta identifica problemas de calidad de código, como vulnerabilidades de seguridad, duplicación de código, y malas prácticas. También proporciona un dashboard con información detallada sobre el estado del código. La empresa dispone de un servidor interno de SonarQube que se puede integrar con el CI. Cada vez que se hace un cambio en el repositorio

de GitLab, se realiza una revisión automática de código. A partir de la información que proporciona, se procede a una fase de refactorización del código. Durante la misma se decide qué bugs, vulnerabilidades y *code smells* se van a corregir. Los bugs afectan directamente a la ejecución, por lo que es conveniente arreglarlos, así como las vulnerabilidades porque comprometen la seguridad.

Los principales bugs resueltos son los relacionados con la gestión de valores nulos. Para resolverlos, se gestionan estos valores a través de excepciones o usando los **Java Optional**. Este concepto hace referencia a una variable que puede tener un valor asignado o que puede contener un valor null. De esa forma con una estructura if podemos acceder al valor del Optional, pero primero se pregunta por él utilizando el método *isPresent()*. Este método nos devuelve true o false dependiendo si el Optional contiene un valor o esta vacío.

En cuanto a los *code smells*, se ha optado por corregir los más críticos y los bloqueantes. Algunos de los problemas estaban relacionados con la complejidad cognitiva de partes del código. Es una medida de cómo es de difícil entender intuitivamente un bloque de código. Para evitar esta complejidad se han añadido nuevos métodos, reduciendo la complejidad de los existentes. También se añaden nuevas clases, para que estén enfocadas en una sola funcionalidad.

También se ha analizado cómo se puede mejorar el rendimiento del sistema, detectando las partes en las que las llamadas consumen más tiempo. Una parte que consume más tiempo es la consulta al catálogo, y como esta consulta se realiza dos veces por cada workflow (primero para generar los ficheros de parámetros para la descarga de datos, y segundo para hacer la estimación del tamaño del volumen de datos), el rendimiento del sistema disminuye. La solución para este tipo de casos ha sido aprovechar la abstracción de caché que ofrece Spring. Es un mecanismo que permite el uso consistente de varios métodos de almacenamiento en caché con un impacto mínimo en el código. De esta manera hacemos que el acceso a los datos sea más rápido y menos costoso, puesto que el acceso a los datos desde la memoria siempre es más rápido en comparación con la obtención de datos desde la base de datos[3].

Una librería de caché para Java es **Caffeine Cache**[11]. Se trata de una librería de alto rendimiento, y proporciona una caché que almacena datos en memoria RAM en lugar de en disco. Una de sus ventajas es que es configurable, pudiendo establecer el tiempo de expiración de las entradas, el tamaño de la propia caché o la política de eliminación para seleccionar qué entradas eliminar cuando se alcance el tamaño máximo.

**Código 11: Uso de Caffeine Cache**

```
@Cacheable(value = CACHE_NAME, key = "{#root.methodName, #step}")
public List<CatalogChunk> getChunks(CatalogDatasourceStep step) {
    var catalogDefinitions = getCatalogDefinitions(step);
    var chunkQuerys = getChunkQueries(catalogDefinitions);
    var uri = URI.create(catalogUrl);
    var chunks = queryCatalog(chunkQuerys, uri);
    return chunks;
}
```

En el fragmento de código anterior se indica con la anotación `@Cacheable` que el resultado de la llamada al método se almacene en la caché especificada. Cuando se llame al método, primero se comprueba la caché, y si ya existe un resultado en caché para ese parámetro, se devuelve directamente el resultado sin ejecutar el método.

# 7. Evaluación y pruebas

La fase de pruebas es una parte fundamental del ciclo de vida del software. Sirve para validar que el software se comporta de acuerdo a los requisitos definidos, y también para identificar y corregir defectos antes de que el software se implemente en producción.

## 7.1. Pruebas unitarias y de integración

Las pruebas unitarias sirven para comprobar que las unidades más pequeñas del software funcionan correctamente. Para realizar las pruebas unitarias se ha utilizado **JUnit**, una biblioteca de pruebas unitarias ampliamente utilizada en el mundo Java. En estas pruebas no se incluyen las interacciones entre diferentes componentes. Como la mayoría de las funcionalidades trabajan a partir de ficheros JSON o YAML obtenidos de servicios externos, se puede simular su comportamiento a partir de ficheros de ejemplo almacenados en `/src/test/resources`.

Las pruebas unitarias han abarcado las siguientes partes:

- Mapeo de JSON con la definición del experimento
- Generadores de tareas (tareas, plantillas de contenedores, dependencias entre tareas)
- Nombrado de las tareas
- Generación de ficheros de parámetros
- Obtención del estado de un workflow
- Obtención del estado de las tareas de un workflow
- Generación del *Request* en JSON que se envía al servidor de Argo

Por otro lado, se realizaron pruebas de integración. Su objetivo es asegurarse de que los diferentes componentes se comportan de forma correcta cuando son puestos a funcionar juntos. En este caso es más sencillo detectar dónde puede haber errores, puesto que el *runner* (motor de cómputo) implementado se prueba junto con otros componentes ya desarrollados y que ya han sido probados. En el clúster Kubernetes local creado para el desarrollo se han desplegado componentes que interaccionan con el componente desarrollado. Estos son: base de datos del catálogo, API de acceso a la base de datos del catálogo, servidor Argo y *backend* (interactúa con la API de gestión de experimentos). De esta manera se comprueba que:

1. El *backend* interactúa correctamente con la API que expone el *runner*.
2. El *runner* interactúa de forma correcta con la API de la base de datos del catálogo, obteniendo los *chunks* adecuados a la consulta realizada
3. La petición que se envía a Argo es procesada y se crea un nuevo workflow con su consecuente ejecución.
4. El *workflow* termina su ejecución sin errores. De esta forma se comprueba que los contenedores encargados de realizar las tareas disponen de toda la información necesaria para realizarla sin errores.
5. El resultado de la ejecución de un *workflow* coincide con el resultado que se obtiene con la anterior implementación antes de realizar la migración a Kubernetes. Los resultados tienen que coincidir si la definición del experimento es la misma.

## 7.2. Pruebas de sistema

Para verificar el comportamiento del sistema en su conjunto se realizaron pruebas de sistema. Las pruebas de sistema se realizaron ya en un entorno de pre-producción. En esta fase ya se puede probar y validar el software en un entorno que simula el entorno de producción. Aquí ya se pueden probar los requisitos no funcionales, comprobando que se guardan los logs de las tareas, que el sistema escala de manera efectiva y que el sistema maneja la carga prevista en producción.

Se han realizado experimentos de mayor volumen, que han detectado la necesidad de utilizar un número mayor de recursos. Por tanto, ha sido necesario ajustar los límites de CPU y memoria asignados a los contenedores que realizan las tareas. También se ha comprobado la eficiencia del sistema, verificando que se eliminan los pods y los volúmenes efímeros al finalizar la ejecución de un workflow.

## 7.3. Pruebas de aceptación

Las pruebas de aceptación se realizan con el sistema completo y finalizado, tras realizar todas las pruebas mencionadas anteriormente. Se realizan para determinar si el software cumple con las necesidades del usuario final, no se pretende descubrir nuevos defectos.

El proceso constó de dos fases. Se pidió a un grupo de desarrolladores que lanzasen varios experimentos. Estos desarrolladores han participado en la implementación de otros componentes del servicio, por lo que conocen detalles científicos y han probado todas las variantes posibles que hay en un experimento. Tras esto, se celebró una reunión en la que no se indicó ninguna incidencia, por lo que se pasó a la siguiente parte.

En esta última fase, se pidió a usuarios habituales del servicio que probasen el nuevo sistema con diversos experimentos. Al finalizar, se pidieron sus opiniones y se transmitió que la experiencia había sido buena. Una vez que el cliente estuvo satisfecho con el nuevo sistema, se concluyó la fase de pruebas.

# 8. Despliegue

Durante la fase de desarrollo se han creado los *manifests* de los recursos Kubernetes de forma manual, ya que al comienzo no se tiene el conocimiento de todos los recursos que se van a necesitar. Sin embargo, el despliegue se realiza con Helm, aprovechando los *charts* ya existentes en el repositorio de la empresa.

El despliegue se realiza en un clúster Kubernetes ya existente, por lo que es necesario tener acceso al mismo. En el archivo `config` en la carpeta `.kube` se almacena la configuración para acceder a un clúster, incluyendo la información de autenticación y los detalles de conexión. El fichero sigue un formato YAML e incluye las siguientes secciones[15]:

1. *clusters*: Detalles como el nombre, la dirección del clúster y la ubicación del archivo de certificado. Se pueden tener múltiples clústeres configurados.

2. *users*: En esta sección se definen los usuarios y sus credenciales para autenticarse en el clúster.

3. *contexts*: Esta sección asocia un clúster y un usuario para formar un contexto. Un contexto define el entorno de trabajo actual, es decir, qué clúster y qué usuario se utilizarán al ejecutar comandos de kubectl.

De esta manera, con `kubectl` se puede acceder a este clúster y gestionarlo, y cambiar entre contextos según sea necesario.

Por otro lado, es necesario tener instalada la herramienta Helm. Helm utiliza kubectl como herramienta subyacente para interactuar con el clúster. Cuando se instala o actualiza un chart, Helm traduce los archivos YAML en comandos de kubectl para crear o modificar los recursos correspondientes. Por tanto, a través de Helm se estarán introduciendo recursos en el clúster de producción.

Antes de comenzar a desplegar los diferentes componentes, en el clúster se crearon dos *namespaces*: **argo** y **climadjust**. El espacio de nombres argo contendrá el servidor Argo, mientras que en climadjust se crean todos los recursos necesarios para desplegar la aplicación.

En primer lugar, se desplegó la base de datos del catálogo. Esto se realizó con un chart de PostgreSQL. El chart proporciona archivos YAML que describen los recursos de Kubernetes necesarios para ejecutar una instancia de PostgreSQL, esos son: secret con el usuario y contraseña del usuario, deployment, service y volume claim. En un fichero de valores se indican los valores por defecto que se quieren sobreescribir. Este fichero incluye valores para la imagen a utilizar, puesto que el catálogo contiene tipos de datos espaciales. La imagen que hay que utilizar es de **PostGis**[4], un sistema de administración de bases de datos PostgreSQL en una base de datos espacial mediante la adición de tres características: tipos de datos espaciales, índices espaciales y funciones que operan sobre ellos. El fichero de valores también incluye valores como los recursos de memoria solicitados o información de acceso (usuario y contraseña).

En segundo lugar, se desplegó la API para acceder a la base de datos del catálogo, con un chart web. Este chart incluye archivos para los siguientes recursos: deployment, service, config map con información de configuración del S3 y un secret con las claves del S3. Como valores a sobreescribir están: imagen de la aplicación web, el puerto interno y la ruta del volumen de datos.

En siguiente lugar, se pasó a desplegar el *runner* implementado y el *backend*. Como son aplicaciones web, también se utiliza el chart web. La información que se le pasa es el nombre de la

imagen a utilizar, el número de puerto interno o el punto de montaje del volumen de datos.

Por último, para desplegar el servidor de Argo se puede utilizar el chart oficial[9]. El chart incluye ficheros para definir el recurso Kubernetes `Workflow`, y para creación de roles, servicios y deployments necesarios en Argo. El siguiente fichero muestra los valores por defecto a sobreescribir.

**Código 12: Valores de configuración para la instalación del chart de Argo**

```
workflow:
  serviceAccount:
    create: true
    name: "argo-workflow"
  rbac:
    create: true
controller:
  workflowNamespaces:
    - default
    - climadjust
```

En el anterior fichero se ha indicado que hay que crear un *service account* (identidad utilizada por los pods para acceder a recursos y realizar operaciones en el clúster), utilizar el control de acceso basado en roles y en qué espacios de nombres se pueden ejecutar *workflows*.

# 9. Conclusiones y trabajos futuros

## 9.1. Conclusiones

El objetivo principal del proyecto era migrar el servicio de ajuste de sesgos climáticos a un clúster Kubernetes. Para ello, se ha utilizado una arquitectura basada en servicios (componentes) independientes y tecnologías modernas como Argo Workflows.

El proyecto se ha dividido en varias fases: análisis de requisitos, implementación, pruebas y despliegue. Durante la fase de análisis se identificaron las necesidades que tendría que cubrir el sistema y se diseñó su arquitectura. La fase de implementación incluyó el desarrollo de los dos componentes: **Java Downloader** (componente para descarga de datos del servicio de almacenamiento en la nube) y **Argo Runner** (motor de cómputo encargado de planificar tareas, gestionar experimentos, etc.). La fase de pruebas implicó elaborar test unitarios y verificar el correcto funcionamiento de la aplicación. Las pruebas unitarias se realizaron en el entorno de desarrollo, mientras que para realizar las pruebas de rendimiento se preparó un entorno de preproducción con la ayuda de Helm.

La arquitectura y diseño escogidos hacen que el sistema sea muy modular y extensible, lo que hace que sea sencillo añadir nuevos módulos o funcionalidades. Esto es gracias a que los diferentes servicios que componen la aplicación se comunican a través de APIs HTTP.

El hecho de haber utilizado Kubernetes como orquestador de contenedores ha proporcionado numerosas ventajas, como la escalabilidad, la alta disponibilidad o la tolerancia a fallos. Argo Workflows se ha beneficiado de todas estas características a la hora de ejecutar los flujos de trabajo. Puede aprovechar la escalabilidad para ejecutar flujos de trabajo de cualquier tamaño y también puede ser utilizado en cualquier plataforma Kubernetes, ya sea local o en la nube.

Tras la finalización del proyecto, se han logrado los objetivos planteados al principio del documento. Además, se han adquirido nuevos conocimientos sobre las tecnologías empleadas que servirán para trabajos futuros.

## 9.2. Trabajos futuros

Durante el desarrollo del proyecto se han ido descubriendo otras funcionalidades en las tecnologías empleadas que serán de utilidad en futuros proyectos. La mayoría están relacionadas con el motor de flujos de trabajo utilizado, Argo Workflows.

En primer lugar, el hecho de haber desplegado un servidor Argo en el clúster Kubernetes, hace que se pueda utilizar para otras aplicaciones cuya ejecución se base en flujos de trabajo. El controlador Argo es capaz de atender varias peticiones a la vez, por lo que sí se podría utilizar el mismo servidor para distintas aplicaciones.

En segundo lugar, se ha establecido un repositorio de artefactos, para guardar los artefactos generados por las tareas. Este repositorio es el configurado en el *config map* del controlador de Argo, por lo que es común para todos los flujos de trabajo. Sin embargo, se podrían configurar varios repositorios y decidir en cada flujo de trabajo cuál va a ser el repositorio utilizado, permitiendo utilizar distintos servicios de almacenamiento.

Al mismo tiempo, se pueden aprovechar los `WorkflowTemplate` que proporciona Argo. Un WorkflowTemplate es una plantilla predefinida que define un conjunto de pasos, parámetros y variables de entorno, y se puede reutilizar para crear flujos de trabajo personalizados. Permite crear una

biblioteca de plantillas de uso frecuente y reutilizarlas haciendo referencia a ellas desde los flujos de trabajo. Esta característica facilitaría la reutilización, ya que en nuevas aplicaciones no sería necesario establecer de nuevo los valores de los metadatos o la especificación del workflow.

Por otro lado, otra característica a incorporar en los flujos de trabajo es el **Exit Handler**. Es una plantilla que siempre se ejecuta, independientemente de si ha acabado con éxito o no. Un caso de uso sería el envío de notificaciones del estado del workflow (por ejemplo, e-mail/Slack).

En siguiente lugar, sería adecuado disponer de alguna herramienta de entrega progresiva. **Argo Rollouts** [7] es un controlador de Kubernetes y un conjunto de CRDs que proporcionan capacidades de implementación avanzada, como experimentación y entrega progresiva a Kubernetes. En entornos de producción de alto volumen a gran escala, una actualización gradual puede ser un procedimiento arriesgado, ya que no se tiene el control sobre los efectos que va a tener en el sistema y no proporciona una revisión automática en caso de fallos. El controlador de Argo Rollouts permite la implementación de diferentes estrategias de despliegue, como por ejemplo:

- *Canary*: Consiste en desplegar la nueva versión en un subconjunto de usuarios para probar su funcionalidad y estabilidad antes de hacer el despliegue completo.
- *Blue/Green*: Consiste en tener dos entornos (cada uno con una versión diferente del software). La versión "azul" es la que se utiliza actualmente, y la "verde" es la nueva versión, en la que se realiza una prueba exhaustiva. Una vez que se verifica que la nueva versión es estable y funciona correctamente, el tráfico se dirige hacia el entorno "verde".
- *AB testing*: Implica crear dos o más versiones del software y proporcionar diferentes versiones a diferentes grupos de usuarios. Se hace un análisis de la respuesta de los diferentes grupos, y se decide qué versión se implementa.

Por ejemplo, en el futuro se puede querer añadir nuevas funcionalidades al servicio de ajuste de sesgos, por lo que existirá una nueva versión. Un usuario puede querer dar progresivamente más tráfico de producción a la nueva versión. Comienza dándole un pequeño porcentaje del tráfico y esperan un tiempo antes de darle más tráfico a la nueva versión. En algún momento, la nueva versión acaba recibiendo todo el tráfico. Con la estrategia *Canary*, el usuario especificaría los porcentajes que desea que reciba la nueva versión y el tiempo de espera entre porcentajes.

Por último, señalar que la clase **ArgoClient.java** implementada para interactuar con el servidor Argo, se puede reutilizar en otros proyectos. La estrategia adecuada sería empaquetar el código desarrollado en una librería y subirla a un repositorio. De esta manera, estará disponible para otros proyectos.

## Referencias


[1] *¿Qué es Docker?* https://aws.amazon.com/es/docker/.

[2] *¿Qué es Kubernetes?* https://kubernetes.io/es/docs/concepts/overview/what-is-kubernetes/.

[3] *A Guide To Caching in Spring*. https://www.baeldung.com/spring-cache-tutorial.

[4] *About PostGIS*. https://postgis.net/.

[5] *Amazon S3*. https://aws.amazon.com/es/s3/.

[6] *Apache Maven Project*. https://maven.apache.org/.

[7] *Argo Rollouts - Kubernetes Progressive Delivery Controller*. https://argo-rollouts.readthedocs.io/en/stable/.

[8] *Argo Workflows - The workflow engine for Kubernetes*. https://argoproj.github.io/argo-workflows/.

[9] *Argo Workflows Chart*. https://github.com/argoproj/argo-helm/tree/main/charts/argo-workflows.

[10] *Artifact Hub*. https://artifacthub.io/.

[11] *Caffeine Cache*. https://github.com/ben-manes/caffeine.

[12] *Climadjust*. https://climadjust.com.

[13] *Climate Data Store*. https://cds.climate.copernicus.eu.

[14] *Componentes de Kubernetes*. https://kubernetes.io/es/docs/concepts/overview/components/.

[15] *Configure Access to Multiple Clusters*. https://kubernetes.io/docs/tasks/access-application-cluster/configure-access-multiple-clusters/.

[16] *Docker Hub*. https://hub.docker.com/.

[17] *Docker: Accelerated, Containeraized Application Development*. https://www.docker.com/.

[18] *GitLab Runner*. https://docs.gitlab.com/runner/.

[19] *Helm Docs*. https://helm.sh/docs/.

[20] M. Iturbide et al. "The R-based climate4R open framework for reproducible climate data access and post-processing". En: *Environmental Modelling Software* 111 (2019), págs. 42-54. ISSN: 1364-8152. DOI: https://doi.org/10.1016/j.envsoft.2018.09.009. URL: https://www.sciencedirect.com/science/article/pii/S1364815218303049.

[21] *Jackson Project Home*. https://github.com/FasterXML/jackson.

[22] *K3s*. https://k3s.io/.

[23] *Longhorn*. https://longhorn.io/.

[24] *Rclone*. https://rclone.org/.

[25] *SonarQube Documentation*. https://docs.sonarqube.org/latest/.

[26] *Spring Framework*. https://spring.io/.

[27] Jean-Noel Thepaut y Dick Dee. "The Copernicus Climate Change Service (C3S): Open Access to a Climate Data Store". En: *EGU General Assembly Conference Abstracts*. EGU General Assembly Conference Abstracts. Abr. de 2016, EPSC2016-9826, EPSC2016-9826.