Autor

Miquel Álvarez Ruiz

Dirigido por

Pedro García López

TRABAJO DE FIN DE GRADO

Optimización del Análisis Genómico mediante la Integración de Lithops en Nextflow

INGENIERÍA INFORMÁTICA

UNIVERSITAT ROVIRA i VIRGILI
Escola Tècnica
Superior d'Enginyeria

Tarragona, 2023

Resumen

Este trabajo se centra en el estudio de Nextflow, una popular herramienta de flujos de trabajo diseñada para facilitar el procesamiento de datos a gran escala en entornos de computación distribuida. Nextflow se utiliza ampliamente en la comunidad científica y de investigación para orquestar flujos de trabajo complejos, proporcionando una manera fácil de definir y ejecutar tareas en paralelo. Su capacidad para trabajar en entornos de computación en la nube hace que sea especialmente relevante en la actualidad, ya que el cloud computing se ha convertido en una solución popular para procesar grandes volúmenes de datos de manera eficiente.

El objetivo principal del estudio fue desarrollar un plugin personalizado para Nextflow que utilizara Lithops como ejecutor de tareas. Lithops es una plataforma sin servidor que permite ejecutar código en entornos cloud, lo que puede brindar beneficios en términos de escalabilidad y eficiencia en el procesamiento de datos genómicos a gran escala. Ello permite a los usuarios aprovechar la infraestructura sin servidor para ejecutar tareas de pipeline de manera distribuida y paralela.

En conclusión, este estudio ofrece una contribución importante al campo de la genómica al extender las capacidades de Nextflow mediante la integración de Lithops.

Resum

Aquest treball se centra en l'estudi de Nextflow, una popular eina de fluxos de treball dissenyada per a facilitar el processament de dades a gran escala en entorns de computació distribuïda. Nextflow s'utilitza àmpliament en la comunitat científica i de recerca per a orquestrar fluxos de treball complexos, proporcionant una manera fàcil de definir i executar tasques en paral·lel. La seva capacitat per a treballar en entorns de computació en el núvol fa que sigui especialment rellevant en l'actualitat, ja que el cloud computing s'ha convertit en una solució popular per a processar grans volums de dades de manera eficient.

L'objectiu principal de l'estudi va ser desenvolupar un plugin personalitzat per a Nextflow que utilitzés Lithops com a executor de tasques. Lithops és una plataforma sense servidor que permet executar codi en entorns cloud, la qual cosa pot brindar beneficis en termes d'escalabilitat i eficiència en el processament de dades genòmiques a gran escala. Això

permet als usuaris aprofitar la infraestructura sense servidor per a executar tasques de pipeline de manera distribuïda i paral·lela.

En conclusió, aquest estudi ofereix una contribució important al camp de la genòmica en estendre les capacitats de Nextflow mitjançant la integració de Lithops.

Abstract

This work focuses on the study of Nextflow, a popular workflow tool designed to facilitate large-scale data processing in distributed computing environments. Nextflow is widely used in the scientific and research community to orchestrate complex workflows, providing an easy way to define and execute tasks in parallel. Its ability to work in cloud computing environments makes it especially relevant today, as cloud computing has become a popular solution for processing large volumes of data efficiently.

The main objective of the study was to develop a custom plugin for Nextflow that uses Lithops as a task runner. Lithops is a serverless platform that allows code to run in cloud environments, which can provide benefits in terms of scalability and efficiency in large-scale genomic data processing. This allows users to take advantage of the serverless infrastructure to execute pipeline tasks in a distributed and parallel manner.

In conclusion, this study offers an important contribution to the field of genomics by extending the capabilities of Nextflow through the integration of Lithops.

Índice

Índice de figuras 5

1. Introducción 7

1.1. Objetivos y motivación del estudio 7

2. Antecedentes 10

2.1. Flujos de trabajo y procesamiento de datos a gran escala 10

2.2. Nextflow: Una herramienta para la administración de flujos de trabajo 11

2.2.1. Executors 11

2.2.2. Canales y procesos 12

2.2.3. Capa FUSE 13

2.2.4. Paralelismo 13

2.2.5. Ficheros usados para el proyecto 14

2.2.6. Scripts 15

2.2.7. Ejecución en el cloud 18

2.2.8. nf-core 21

2.3. Cloud Computing y su aplicación en el procesamiento de datos 21

2.3.1. Object Storage 21

2.3.2. Plataformas de servicios en la nube 21

2.4. Lithops: Una plataforma para el procesamiento distribuido en la nube 22

2.4.1. Configuración de ejecución 23

2.4.2. Integración con otros servicios 24

2.4.3. Estructura de funciones 24

2.4.4. Empaquetado de dependencias 24

2.4.5. Almacenamiento y comunicación 25

2.4.6. Ejemplo script Lithops 25

3. Metodología 27

3.1. Modelo Cascada para el Desarrollo del Prototipo 27

3.1.1. Requisitos 28

3.1.2. Diseño 28

3.1.3. Implementación 29

3.1.4. Pruebas 30

3.2. Implementación 30

3.2.1.Generación ficheros y petición 30

3.2.2.Lógica de Flask 35

3.2.3.Ejecución en Lithops 36

3.3. Entorno de ejecución **40**
4. Validación 42
4.1. Tiempos de ejecución **42**
4.2. Costes de ejecución **45**
4.3. Facilidad de uso **47**
4.4. Valoración final **49**
5. Futuro del plugin 51
6. Conclusiones 53
Referencias 54
Anexo 55
A Ficheros carga y descarga para pruebas individuales 55
A.1. lithops_uploader.py **55**
A.2 lithops_downloader.py **55**
B Servidor flask 57
B.1. flask_app.py **57**

Índice de figuras

Figura 1: Logotipo de nextflow 11
Figura 2: Ejecución de Nextflow en AWS Batch 20
Figura 3: Ejecución de Nextflow en MS Azure 20
Figura 4: Logotipo de nf-core 21
Figura 5: Logotipo de AWS 21
Figura 6: Logotipo de MS Azure 21
Figura 7: Logotipo de Google Cloud 21
Figura 8: Logotipo de Lithops 22
Figura 9: Diagrama de secuencias 29
Figura 10: Diagrama ejecución LithFlow 30
Figura 11: Ejemplo estructura ficheros work 31
Figura 12: Execution timelapse AWS 43
Figura 13: Execution timelapse Lithops 44

Índice de código

Código 1: Pipeline Básica 17
Código 2: Function executor Lithops 25
Código 3: Código base de .command.run 32
Código 4: nxf_launch() modificado 33
Código 5: Porción del fichero “flask_app.py” 35
Código 6: docker_func() 39
Código 7: script_func() 38
Código 8: lithops_uploader.py 55
Código 9: lithops_downloader.py 56
Código 10: flask_app.py 62

1. Introducción

La gestión eficiente de flujos de trabajo y el procesamiento de grandes volúmenes de datos se han vuelto fundamentales en diversos campos, como la investigación científica, la bioinformática y la ciencia de datos. Para abordar esta necesidad, se han desarrollado herramientas especializadas que facilitan la definición, ejecución y supervisión de flujos de trabajo complejos. Una de estas herramientas ampliamente utilizada es Nextflow [1].

Nextflow es una popular herramienta de flujos de trabajo diseñada para facilitar el procesamiento de datos a gran escala en entornos de computación distribuida. Permite a los investigadores y científicos orquestar flujos de trabajo complejos, definiendo tareas y especificando cómo se deben ejecutar en paralelo. Su lenguaje de dominio específico (DSL) intuitivo y flexible lo convierte en una elección preferida en la comunidad científica e investigadora.

En la actualidad, el cloud computing se ha convertido en una solución cada vez más popular para el procesamiento eficiente de grandes volúmenes de datos. Las infraestructuras en la nube ofrecen escalabilidad, flexibilidad y recursos bajo demanda, lo que permite ejecutar flujos de trabajo distribuidos de manera eficiente. La capacidad de Nextflow para trabajar en entornos de computación en la nube lo hace especialmente relevante en este contexto.

1.1. Objetivos y motivación del estudio

El objetivo principal de este trabajo es desarrollar un plugin personalizado para Nextflow que utilice Lithops [2] como ejecutor de tareas. Lithops es una plataforma sin servidor que permite ejecutar código en entornos cloud, brindando beneficios en términos de escalabilidad y eficiencia en el procesamiento de datos genómicos a gran escala. La integración de Lithops en Nextflow ampliará las capacidades de esta herramienta, permitiendo a los usuarios aprovechar la infraestructura sin servidor para ejecutar tareas de pipeline de manera distribuida y paralela.

Los objetivos planteados para este proyecto son los siguientes:

- Integrar Lithops con Nextflow para aprovechar las capacidades de procesamiento de datos de Lithops.
- Desarrollar un plugin funcional que permita la ejecución de tareas de Nextflow utilizando Lithops como ejecutor.
- Investigar y comprender en profundidad los entornos de Nextflow y Lithops para garantizar una integración adecuada.
- Evaluar el rendimiento del plugin desarrollado comparándolo con otros ejecutores de tareas disponibles en Nextflow.
- Analizar los resultados obtenidos y discutir las ventajas y desafíos de utilizar Lithops como ejecutor en el contexto de Nextflow.
- Demostrar la viabilidad de la integración entre Nextflow y Lithops a través del desarrollo de un prototipo funcional.
- Explorar posibles aplicaciones y futuras mejoras en el uso de Lithops como ejecutor en el entorno de Nextflow.
- Evaluar el uso del plugin en una pipeline real de genómica, comparando la sencillez de la infraestructura y la eficiencia en el procesamiento de datos genómicos, centrándose en la facilidad de uso para el usuario.
- Contribuir a la comunidad científica y de desarrollo de Nextflow mediante la presentación del plugin desarrollado y los resultados obtenidos.

Con estos objetivos, se busca no solo facilitar la ejecución eficiente de pipelines en Nextflow, sino también abrir nuevas posibilidades en el procesamiento de datos genómicos a gran escala mediante la utilización de la infraestructura sin servidor proporcionada por Lithops.

Antes de iniciar el desarrollo del plugin, fue necesario adquirir familiaridad con los entornos de Nextflow y Lithops para comprender su funcionamiento y sus posibilidades, ya que no tenía experiencia previa con ninguno de ellos.

Una de las principales dificultades en la integración fue la falta de documentación por parte de Nextflow sobre cómo crear un ejecutor de tareas personalizado. Solo se proporcionaba un esqueleto básico para la creación de un plugin. Por lo tanto, fue necesario realizar una investigación exhaustiva del código fuente de Nextflow disponible en su repositorio de GitHub para comprenderlo en profundidad antes de comenzar a escribir el código del plugin.

Además, se realizará una evaluación del rendimiento del plugin desarrollado, comparándolo con otros ejecutores de tareas disponibles en Nextflow. Se analizarán los resultados obtenidos y se discutirán las ventajas, desafíos y posibles aplicaciones del uso de Lithops como ejecutor dentro del contexto de Nextflow.

La motivación de este proyecto radica en la necesidad de mejorar la eficiencia y escalabilidad del procesamiento de datos en flujos de trabajo a gran escala. La combinación de Nextflow y Lithops tiene el potencial de ofrecer una solución más eficiente y flexible para los investigadores y científicos que trabajan con grandes volúmenes de datos genómicos, lo que puede impulsar el avance de la investigación en este campo y facilitar el descubrimiento de nuevos conocimientos y avances científicos.

El proyecto se ha llevado a cabo en el entorno de CloudLab [3], un grupo de investigación de la Universidad Rovira i Virgili (URV). Dentro del cual agradezco sinceramente a Xavier Roca Canals por su dedicación al brindar su tiempo en introducirme y ayudarme con la configuración del entorno de Lithops y brindarme conocimientos básicos sobre genómica y sobre FASTA y FASTQ. Ficheros ampliamente usados en el mundo de la genómica detallados más adelante en este proyecto.

2. Antecedentes

Esta sección se centrará en los flujos de trabajo y el procesamiento de datos a gran escala. Se explorará Nextflow como una herramienta para la administración de flujos de trabajo, cubriendo aspectos como los ejecutores, los canales y procesos y el paralelismo. También se mencionarán los archivos, scripts y la ejecución en la nube, incluyendo nf-core. Además, se tratará el Cloud Computing y su aplicación en el procesamiento de datos, con un enfoque en el almacenamiento de objetos y las plataformas de servicios en la nube. Por último, se presentará Lithops como una plataforma para el procesamiento distribuido en la nube, describiendo su configuración de ejecución, integración con otros servicios, estructura de funciones, empaquetado de dependencias y almacenamiento y comunicación.

2.1. Flujos de trabajo y procesamiento de datos a gran escala

En los últimos años, la cantidad de datos generados en diferentes áreas, como la genómica, la astronomía y la investigación biomédica, ha experimentado un crecimiento exponencial. Estos grandes volúmenes de datos requieren soluciones de procesamiento eficientes y escalables.

Los flujos de trabajo se utilizan para organizar y automatizar la ejecución de múltiples tareas interdependientes. Permiten definir la secuencia y la lógica de las operaciones, gestionar la transferencia de datos entre etapas y controlar la ejecución paralela de tareas.

Para abordar estos desafíos, han surgido herramientas especializadas en la administración de flujos de trabajo, como Nextflow. Nextflow es una popular herramienta que ofrece una forma sencilla y declarativa de definir y ejecutar flujos de trabajo en entornos distribuidos y paralelos. Proporciona una sintaxis intuitiva y flexible para describir los componentes del flujo de trabajo, así como para manejar la escalabilidad y la gestión de recursos en la nube.

Integrando soluciones de computación en la nube, como el cloud computing, en los flujos de trabajo, se pueden aprovechar los beneficios de escalabilidad, elasticidad y costo-efectividad que ofrece este entorno. El cloud computing ha revolucionado la forma en

que se procesan y almacenan los datos, permitiendo a los investigadores escalar sus recursos de manera eficiente según las necesidades del análisis.

En conclusión, los flujos de trabajo y el procesamiento de datos a gran escala son áreas de investigación en constante evolución. Herramientas como Nextflow, combinadas con el poder del cloud computing, ofrecen soluciones cada vez más robustas para gestionar y analizar grandes volúmenes de datos de manera eficiente y escalable.

2.2. Nextflow: Una herramienta para la administración de flujos de trabajo

Nextflow es una herramienta escrita en Groovy de administración de flujos de trabajo desarrollada específicamente para abordar los desafíos del procesamiento de datos a gran escala en entornos distribuidos. Proporciona un enfoque declarativo para definir flujos de trabajo complejos y paralelos, permitiendo a los usuarios describir las dependencias entre tareas y especificar los recursos necesarios para su ejecución.

Figura 1. Logotipo de nextflow

Nextflow utiliza una sintaxis basada en un lenguaje de programación específico del dominio (DSL), que simplifica la definición y el control de flujos de trabajo. Además, Nextflow está diseñado para ser altamente portable, lo que significa que puede ejecutarse en diversas infraestructuras, incluyendo sistemas locales, clústeres de computación de alto rendimiento y entornos de nube.

2.2.1. ***Executors***

Nextflow ofrece diferentes executors, que son los responsables de ejecutar las tareas definidas en el flujo de trabajo. Algunos de los executors disponibles son:

- Executor local: Permite ejecutar tareas en la máquina local donde se ejecuta Nextflow. Es útil para pruebas y desarrollo en entornos de una sola máquina.

- Clúster de computación: Nextflow es compatible con una amplia variedad de sistemas de administración de trabajos en clústeres, como PBS, SGE, SLURM y LSF. Esto permite aprovechar los recursos de un clúster de computación para realizar tareas en paralelo y distribuidas en el clúster.
- Cloud: Nextflow también es compatible con entornos de nube pública, como Amazon Web Services (AWS), Microsoft Azure y Google Cloud Platform (GCP). Esto permite aprovechar la escalabilidad y la disponibilidad de los recursos en la nube para ejecutar flujos de trabajo en entornos distribuidos.

2.2.2. Canales y procesos

Los canales son las estructuras de datos que permiten el flujo de información entre las tareas. Los canales actúan como conductos a través de los cuales se transmiten los datos generados por una tarea para ser utilizados como entrada en otra tarea. Los canales pueden ser simples (que transmiten datos en un solo sentido) o múltiples (que transmiten datos en múltiples direcciones).

Otro componente importante son los procesos. Son las unidades de trabajo que realizan tareas específicas en el flujo de trabajo. Cada proceso se define con un bloque de código que contiene el comando o script a ejecutar, así como cualquier entrada y salida de datos asociados. Los procesos pueden tener dependencias entre sí, lo que permite la ejecución secuencial o en paralelo de las tareas.

La interacción entre canales y procesos es uno de los aspectos fundamentales de Nextflow. Los canales permiten la transferencia de datos entre procesos de manera eficiente y flexible, lo que facilita la construcción de flujos de trabajo complejos y la gestión automática de la entrada y salida de datos.

2.2.3. Capa FUSE

Nextflow también utiliza una capa llamada FUSE [8] (Filesystem in Userspace) para facilitar la gestión de los datos en los flujos de trabajo. FUSE es una interfaz que permite a los usuarios crear sistemas de archivos personalizados en el espacio de usuario, lo que significa que los sistemas de archivos pueden ser implementados en programas de nivel de usuario en lugar de requerir privilegios de administrador.

En el contexto de Nextflow, la capa FUSE se utiliza para crear y gestionar de manera transparente los sistemas de archivos virtuales asociados con los flujos de trabajo. Esto permite que los datos se accedan y manipulen como si estuvieran en un sistema de archivos tradicional, a pesar de que pueden estar almacenados en diferentes ubicaciones físicas o en entornos de almacenamiento en la nube.

La capa FUSE en Nextflow se integra con los canales y procesos de manera que los datos fluyen a través de los sistemas de archivos virtuales creados por FUSE. Esto proporciona una abstracción adicional que facilita la gestión y manipulación de los datos de entrada y salida en los flujos de trabajo.

*2.2.4. **Paralelismo***

Nextflow facilita el paralelismo de dos maneras principales:

- Paralelismo entre tareas: Puedes definir dependencias entre tareas en Nextflow y la herramienta se encargará automáticamente de ejecutar tareas independientes en paralelo. Esto significa que las tareas que no tienen dependencias entre sí se pueden ejecutar simultáneamente, aprovechando al máximo los recursos disponibles y acelerando la ejecución del flujo de trabajo.

- Paralelismo dentro de las tareas: Nextflow también permite el paralelismo dentro de las tareas individuales. Puedes especificar configuraciones de recursos para cada tarea, como el número de núcleos de CPU, memoria y otros recursos. Esto permite que las tareas se ejecuten de manera paralela internamente, utilizando eficientemente los recursos disponibles dentro de cada tarea.

El paralelismo en Nextflow se gestiona de manera automática y optimizada, teniendo en cuenta la disponibilidad de recursos y las restricciones definidas. Esto proporciona un rendimiento mejorado y una reducción significativa en el tiempo de ejecución de los flujos de trabajo, especialmente cuando se procesan grandes volúmenes de datos.

El paralelismo en Nextflow es especialmente útil en entornos de computación distribuida, como clústeres y entornos de nube, donde se pueden aprovechar múltiples recursos de manera simultánea para realizar tareas en paralelo.

2.2.5. Ficheros usados para el proyecto

En el estudio de la genómica, existen varios tipos de archivos de datos que son ampliamente utilizados en el procesamiento de flujos de trabajo. Algunos de los tipos de archivo más comunes incluyen:

Ficheros FASTA (FASTA files)

Los ficheros FASTA [10] son una representación común para secuencias biológicas, como secuencias de ADN o proteínas. Estos ficheros contienen una serie de registros, donde cada registro consta de una línea de encabezado que comienza con el carácter ">" seguido de una descripción y una o varias líneas que contienen la secuencia correspondiente. Los ficheros FASTA son utilizados para almacenar y compartir secuencias biológicas en el campo de la bioinformática y son ampliamente utilizados en flujos de trabajo de análisis genómico.

Ficheros FASTQ (FASTQ files)

Los ficheros FASTQ [11] son utilizados para almacenar secuencias biológicas junto con información de calidad asociada. Cada registro en un fichero FASTQ consta de cuatro líneas: la primera línea contiene un identificador de secuencia, la segunda línea contiene la secuencia en sí, la tercera línea comienza con el carácter "+" y puede contener información adicional opcional, y la cuarta línea contiene una cadena de valores de calidad que representan la confiabilidad de cada base de la secuencia. Los ficheros FASTQ son fundamentales en el análisis de datos de secuenciación de próxima generación (NGS), ya que proporcionan información valiosa sobre la calidad de las secuencias obtenidas.

Estos son solo algunos ejemplos de los tipos de archivos de datos utilizados en Nextflow. Dependiendo del campo de estudio y el flujo de trabajo específico, pueden existir otros formatos de archivo utilizados para representar diferentes tipos de datos biológicos o científicos. Nextflow ofrece una flexibilidad significativa para trabajar con diferentes tipos de archivos y proporciona herramientas y funciones para el manejo y procesamiento eficiente de estos datos en flujos de trabajo.

2.2.6. Scripts

Los scripts desempeñan un papel fundamental en la definición y ejecución de los flujos de trabajo. Estos scripts se escriben en un DSL. El DSL [9] es un lenguaje específico de dominio diseñado para describir flujos de trabajo científicos y de procesamiento de datos a gran escala. El DSL de Nextflow está optimizado para definir de manera concisa y legible las tareas, las entradas y salidas, y las relaciones entre ellas (canales) en un flujo de trabajo.

Cada tarea está compuesta por varios elementos clave:

- Input y output: Estas palabras clave se utilizan para declarar las entradas y salidas de una tarea. Mediante la especificación de las entradas, se indica qué datos o archivos necesita la tarea para ejecutarse. Las salidas definen los resultados o archivos generados por la tarea. Estos elementos permiten establecer las

dependencias entre las tareas y garantizar que los datos adecuados estén disponibles antes de iniciar la ejecución. Estos datos, se reciben y se envían por los canales previamente explicados.

- El script de una tarea contiene el código que se ejecutará dentro de la tarea. Puede ser un script de shell, un comando de línea de comandos u otro código en el lenguaje preferido. En Nextflow, el script se define dentro de bloques de código delimitados por triples comillas ("""). Estos bloques contienen el código necesario para realizar las operaciones específicas de la tarea, como transformaciones de datos, análisis o ejecución de herramientas externas.

- El elemento 'params' se utiliza para especificar los parámetros que se pueden configurar al ejecutar el flujo de trabajo, como el archivo de entrada o los valores de configuración.

Por último, en cada fichero de Nextflow de una pipeline, se especifica el término *workflow*. En este contexto, se refiere a la definición global del flujo de trabajo. Es donde se especifican las etapas y las relaciones entre ellas. El bloque workflow en el script de Nextflow engloba todas las etapas y define cómo se conectan entre sí.

El código que se muestra a continuación es un ejemplo de un script / pipeline de Nextflow:

```
1   #!/usr/bin/env nextflow
2   params.in = "$baseDir/data/sample.fa"
3   /*
4    * Split a fasta file into multiple files
5    */
6   process splitSequences {
7       input:
8       path 'input.fa'
9
10      output:
11      path 'seq_*'
12
```

```
13      """
14      awk '/^>/{f="seq_"++d} {print > f}' < input.fa
15      """
16  }
17  /*
18   * Reverse the sequences
19   */
20  process reverse {
21      input:
22      path x
23
24      output:
25      stdout
26
27      """
28      cat $x | rev
29      """
30  }
31  /*
32   * Define the workflow
33   */
34  workflow {
35      splitSequences(params.in) \
36        | reverse \
37        | view
38  }
```

Código 1. Pipeline Básica

En este ejemplo, se define un flujo de trabajo que consta de tres etapas: *splitSequences*, *reverse* y *view*. A continuación, se explica brevemente cada una de ellas:

- splitSequences: Esta etapa se encarga de dividir un archivo fasta en varios archivos más pequeños. Toma como entrada un archivo denominado input.fa y produce varios archivos de salida que comienzan con el prefijo "seq_". Esto se logra mediante el uso de un comando awk que busca líneas que comiencen con el símbolo ">" y crea un nuevo archivo cada vez que se encuentra una línea de este tipo.

- reverse: En esta etapa, se invierten las secuencias del archivo de entrada. Recibe como entrada un archivo denominado x y produce una salida en la consola utilizando el comando rev, que invierte el orden de los caracteres de cada línea.

- view: Esta etapa representa la visualización de los resultados en la consola. No se muestra la implementación exacta de esta etapa en el código proporcionado, pero se puede asumir que es responsable de mostrar los resultados en algún formato legible para el usuario.

2.2.7. Ejecución en el cloud

Durante el desarrollo de este trabajo, se utilizó el executor de AWS Batch como entorno para probar y aprender Nextflow.

En el contexto de Nextflow, el executor de AWS Batch permite enviar trabajos y orquestar la ejecución de tareas en entornos de AWS Batch. Para utilizar este executor, es necesario configurar previamente los componentes necesarios en AWS, como las colas de trabajos y los entornos de computación. Antes doy una breve explicación de estos componentes:

- Las colas de trabajo en AWS Batch son espacios donde se almacenan los trabajos que se enviarán para su ejecución. Cada cola de trabajo tiene una prioridad asignada y está asociada con uno o más entornos de cómputo. Las colas de trabajo son responsables de despachar los trabajos a los entornos de cómputo correspondientes. Los trabajos en una cola se procesan según el orden en que se encuentran en la cola.

- Los entornos de computación son los recursos de procesamiento utilizados por AWS Batch para ejecutar los trabajos. En AWS Batch, existen dos tipos de entornos de cómputo: administrados y no administrados. Los entornos de cómputo

administrados se escalan automáticamente según la demanda de recursos, lo que permite una gestión eficiente y flexible de los recursos de cómputo. Al configurar un entorno de cómputo, se deben especificar detalles como el modelo de aprovisionamiento (por ejemplo, instancias bajo demanda o instancias Spot), el número mínimo y máximo de recursos de cómputo, los tipos de instancias admitidos y otras opciones de configuración.

Al utilizar el executor de AWS Batch en Nextflow, se aprovecha la escalabilidad y la capacidad de administración automática de los recursos de computación proporcionados por AWS. Esto permite que los flujos de trabajo sean distribuidos y ejecutados en paralelo de manera eficiente, optimizando el tiempo de ejecución y la utilización de recursos.

En la Figura 7 se muestra el flujo de ejecución de un pipeline en este entorno. El proceso comienza desde una instancia de EC2 o desde un entorno local, donde Nextflow se ejecuta especificando los parámetros y el pipeline que se desea ejecutar. A continuación, Nextflow lee la configuración del archivo y determina a qué cola deben enviarse las tareas. Una vez que una tarea está lista para ser ejecutada, se envía a una instancia de EC2 correspondiente con los recursos necesarios. Dependiendo de la tarea, esta puede ejecutarse dentro de un contenedor Docker que se descargará en el momento de ejecutar la tarea, o directamente en la instancia si no se requiere algún binario. Tanto las instancias

de EC2 como el nodo principal están conectados a un bucket de S3, donde se almacenan los archivos intermedios y que facilitan la comunicación entre las tareas.

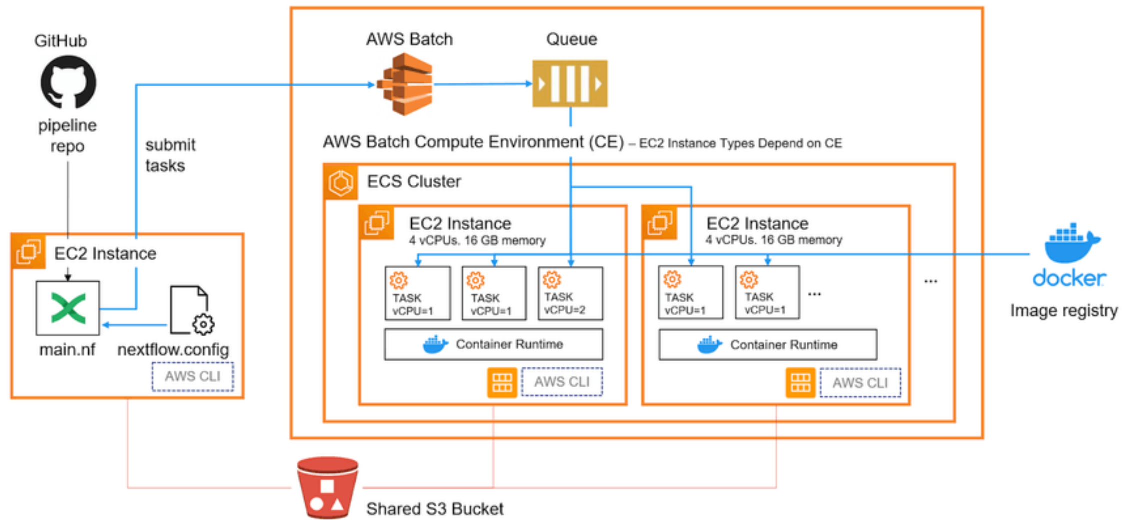

Figura 2. Ejecución de Nextflow en AWS Batch

Además, se realizó un intento de ejecutar Nextflow en Azure. Sin embargo, se encontraron algunas limitaciones en la suscripción de estudiante que dificultaron la ejecución de pipelines más complejas. A pesar de esto, la figura 8 presenta el diagrama de ejecución para este entorno en la nube, el cual guarda similitudes con el de AWS.

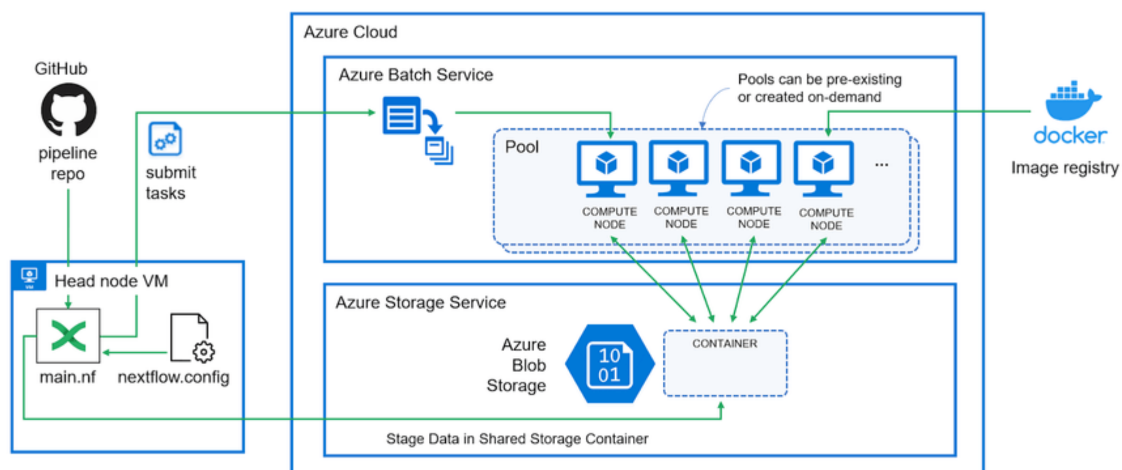

Figura 3: Ejecución de Nextflow en MS Azure

2.2.8. nf-core

Nextflow también se ha beneficiado de la creación y adopción de nf-core [7], una iniciativa de la comunidad científica para desarrollar y mantener flujos de trabajo estándar y reproducibles. nf-core proporciona una colección de flujos de trabajo de Nextflow optimizados para aplicaciones específicas en genómica y biología computacional. Estos flujos de trabajo estandarizados ofrecen una estructura común, buenas prácticas y documentación detallada, lo que facilita a los investigadores la implementación y ejecución de análisis genómicos de manera confiable y reproducible. La comunidad nf-core fomenta la colaboración, revisión y mejora continua de los flujos de trabajo, lo que ha contribuido a la robustez y confiabilidad de Nextflow como herramienta de administración de flujos de trabajo en el ámbito científico y de investigación.

Figura 4. Logotipo de nf-core

2.3.Cloud Computing y su aplicación en el procesamiento de datos

El cloud computing ha transformado la forma en que las organizaciones y los investigadores abordan el procesamiento de grandes volúmenes de datos. Con el cloud computing, los recursos informáticos, como servidores y almacenamiento, se pueden escalar de manera elástica según la demanda, lo que permite procesar grandes volúmenes de datos de manera eficiente y rentable.

En la actualidad, existen varios proveedores de servicios en la nube reconocidos, como Amazon Web Services (AWS) [4], Microsoft Azure [5] y Google Cloud [6].

Figura 5. Logotipo de AWS

Figura 6. Logotipo de MS Azure

Figura 7. Logotipo de Google Cloud

Estos proveedores ofrecen una amplia gama de servicios y herramientas para el procesamiento de datos a gran escala.

2.3.1. Object Storage

Uno de los servicios clave proporcionados por los proveedores de la nube es el almacenamiento de objetos. El almacenamiento de objetos es un método para almacenar y recuperar grandes cantidades de datos no estructurados, como archivos, imágenes y videos. Los objetos almacenados en este tipo de almacenamiento se organizan en contenedores lógicos llamados *buckets* y se accede a ellos a través de una API.

El almacenamiento de objetos ofrece durabilidad, escalabilidad y alta disponibilidad. Los proveedores de la nube ofrecen soluciones de almacenamiento de objetos como Amazon S3 en AWS, Azure Blob Storage en Microsoft Azure y Google Cloud Storage en GCP. Estos servicios permiten a los usuarios almacenar y recuperar datos de manera eficiente, así como aprovechar funciones adicionales, como el cifrado de datos y la replicación en varias regiones geográficas para garantizar la disponibilidad y la durabilidad de los datos.

2.3.2. Plataformas de servicios en la nube

Además del almacenamiento de objetos, los proveedores de servicios en la nube ofrecen una amplia gama de servicios adicionales para el procesamiento de datos. Estas plataformas incluyen:

- Function-as-a-Service (FaaS): Estas plataformas, como AWS Lambda, Azure Functions y Google Cloud Functions, permiten ejecutar funciones de manera eficiente y escalable sin preocuparse por la infraestructura subyacente. Los desarrolladores pueden escribir y cargar funciones individuales que se ejecutan en respuesta a eventos específicos, lo que facilita la creación de aplicaciones basadas en eventos y el procesamiento de datos en tiempo real.
- Batch Processing: Los servicios de procesamiento en lotes, como AWS Batch y Azure Batch, permiten ejecutar trabajos de procesamiento de datos en grandes volúmenes de manera distribuida y escalable. Estos servicios ofrecen la capacidad

de administrar y programar trabajos en lotes, aprovechando la potencia de cómputo de la nube para procesar grandes conjuntos de datos de manera eficiente.

2.4.Lithops: Una plataforma para el procesamiento distribuido en la nube

Lithops es una plataforma sin servidor diseñada específicamente para el procesamiento de datos distribuido en entornos de computación en la nube. Proporciona una capa de abstracción sobre los servicios en la nube, permitiendo a los desarrolladores ejecutar código en paralelo de manera distribuida sin tener que preocuparse por la gestión de la infraestructura subyacente.

Figura 8. Logotipo de Lithops

Lithops es compatible con varios proveedores de servicios en la nube, como AWS, Azure y GCP, lo que te brinda flexibilidad para utilizar los recursos de la nube que mejor se adapten a tus necesidades. Además, Lithops se integra fácilmente con Nextflow, lo que te permite ampliar las capacidades de Nextflow al aprovechar la escalabilidad y eficiencia proporcionadas por la plataforma sin servidor.

Lithops se basa en cinco premisas fundamentales extraídas de su página web:

1. Fácil de usar: Lithops ha sido diseñado para ser intuitivo y accesible para los usuarios. Su interfaz sencilla y amigable facilita su adopción y utilización, permitiendo a los desarrolladores e investigadores aprovechar sus capacidades sin dificultades.

2. Fácil de aprender: Además de ser fácil de usar, Lithops también ofrece una curva de aprendizaje suave. Su documentación completa y ejemplos de uso claros proporcionan a los usuarios los recursos necesarios para comprender rápidamente sus características y funcionalidades. Esto permite que incluso aquellos con poca

experiencia en programación en la nube puedan utilizar Lithops de manera efectiva.

3. Cross-platform: Lithops está diseñado para funcionar de manera transparente en diferentes plataformas. Es compatible con los principales proveedores de servicios en la nube, como AWS, Azure y GCP, lo que permite a los usuarios aprovechar la infraestructura en la nube de su elección sin restricciones. Esta capacidad de operar en diversas plataformas brinda flexibilidad y opciones a los usuarios, adaptándose a sus necesidades y preferencias individuales.

4. Error friendly: Lithops ha sido desarrollado teniendo en cuenta la capacidad de recuperación ante errores. Incluso en situaciones donde pueden ocurrir fallos en la ejecución, Lithops está diseñado para manejar estos errores de manera elegante y proporcionar información detallada sobre los problemas encontrados. Esto ayuda a los usuarios a identificar y solucionar rápidamente cualquier problema, garantizando una ejecución más confiable y eficiente de sus tareas.

5. Escalado dinámico: Una de las características más destacadas de Lithops es su capacidad de escalado dinámico. La plataforma puede adaptarse automáticamente a las demandas de procesamiento, escalando hacia arriba o hacia abajo según sea necesario. Esto permite a los usuarios aprovechar la potencia de cómputo de la nube y realizar el procesamiento de datos a gran escala de manera eficiente y rentable.

2.4.1. Configuración de ejecución

Lithops proporciona opciones de configuración flexibles para adaptarse a los requisitos específicos de cada tarea o flujo de trabajo. Los usuarios pueden especificar el número de workers (trabajadores) que desean utilizar para procesar las tareas, así como la cantidad de memoria asignada a cada worker. Esto permite ajustar el rendimiento y los recursos utilizados de acuerdo con las necesidades de procesamiento de datos. Además, Lithops

proporciona mecanismos para controlar el tiempo de espera de las tareas y definir políticas de reintentos en caso de fallos.

2.4.2. Integración con otros servicios

Lithops se integra estrechamente con otros servicios en la nube, como almacenamiento de objetos y bases de datos. Esto permite a los usuarios acceder y procesar datos almacenados en servicios como Amazon S3, Google Cloud Storage. La integración con estos servicios facilita la lectura y escritura distribuida de datos de gran escala, lo que resulta especialmente útil en flujos de trabajo que requieren el procesamiento de conjuntos de datos extensos.

2.4.3. Estructura de funciones

En Lithops, las tareas se definen como funciones Python. Cada función representa una unidad de trabajo individual que se ejecutará de forma distribuida. Los usuarios pueden definir estas funciones según sus necesidades específicas, utilizando parámetros y variables adecuadas para cada tarea. Lithops permite a los usuarios escribir código personalizado y aprovechar las bibliotecas y módulos de Python que necesiten para realizar sus tareas.

2.4.4. Empaquetado de dependencias

Lithops ofrece la capacidad de empaquetar y gestionar las dependencias y paquetes de Python necesarios para ejecutar las funciones. Los usuarios pueden especificar las dependencias requeridas por sus tareas, como bibliotecas externas o módulos personalizados, y Lithops se encarga de empaquetar estas dependencias junto con las funciones para su correcta ejecución en los workers. Esto asegura que las funciones tengan acceso a todas las bibliotecas y módulos necesarios durante la ejecución distribuida de tareas.

2.4.5. Almacenamiento y comunicación

Lithops utiliza mecanismos de almacenamiento distribuido para guardar los resultados intermedios y comunicarse entre las tareas. Los usuarios pueden especificar un bucket de almacenamiento en la nube (por ejemplo, un bucket de Amazon S3 o un Cloud Storage Bucket de Google Cloud) donde se almacenarán los datos intermedios generados durante la ejecución de las tareas. Esto permite una comunicación eficiente y segura entre las tareas distribuidas y evita la pérdida de datos.

2.4.6. Ejemplo script Lithops

A continuación se muestra un ejemplo de código extraído del repositorio de Lithops que ilustra cómo ejecutar una función de manera distribuida:

```
1  from lithops import FunctionExecutor
2
3  def hello(name):
4      return 'Hello {}!'.format(name)
5
6  with FunctionExecutor() as fexec:
7      fut = fexec.call_async(hello, 'World')
8      print(fut.result())
```

Código 2. Function executor Lithops

En primer lugar, se crea una instancia de FunctionExecutor de Lithops, que actuará como el ejecutor de funciones. Esta instancia nos permite aprovechar la infraestructura cloud para ejecutar nuestras funciones de forma escalable y distribuida.

En el código, se define la función hello, la cual acepta un parámetro de tipo cadena y devuelve un saludo formateado con ese nombre.

Después de crear el FunctionExecutor, se utiliza el método call_async para invocar la función hello. En este caso, se pasa el argumento "World" como nombre.

La llamada a call_async es asíncrona, lo que significa que no bloquea la ejecución del programa. En su lugar, devuelve un objeto "fut", que representa el resultado de la ejecución de la función.

Para obtener el resultado final de la función, se utiliza el método result() del objeto fut. Esto bloquea la ejecución hasta que la función se complete y devuelve el resultado.

Finalmente, el resultado obtenido se muestra por pantalla.

Al ejecutar este código, se aprovechará la configuración previamente establecida en Lithops para determinar el backend de ejecución. Dependiendo de dicha configuración, las funciones se ejecutarán en el backend cloud especificado, lo que nos permite aprovechar la escalabilidad y capacidad de procesamiento de la infraestructura cloud.

3. Metodología

En esta sección, se presentará la metodología utilizada para el desarrollo del proyecto. En primer lugar, se realizará un resumen del enfoque adoptado, que se basó en el modelo cascada para el desarrollo de software. Este modelo proporcionó una estructura sólida y secuencial para abordar las diferentes etapas del proyecto, desde el análisis de requisitos hasta la implementación del prototipo.

Posteriormente, se profundizará en la implementación y ejecución del prototipo, donde se describirán en detalle las actividades y decisiones tomadas durante este proceso. Se proporcionará una visión más profunda del código desarrollado, destacando aspectos clave y soluciones adoptadas para abordar los desafíos específicos del proyecto.

3.1. Modelo Cascada para el Desarrollo del Prototipo

El modelo cascada es un enfoque de desarrollo de software que sigue un flujo lineal y secuencial, dividiendo el proceso en etapas bien definidas. Cada etapa se basa en los resultados y entregables de la etapa anterior, lo que garantiza un desarrollo ordenado y estructurado del prototipo final. A continuación, se detallan las etapas principales del modelo cascada aplicado al desarrollo del prototipo:

3.1.1. ***Requisitos***

Durante esta fase, se lleva a cabo un análisis exhaustivo de los executors existentes de Nextflow, como AWS y Azure, para comprender su funcionamiento y utilizarlos como referencia para el nuevo executor basado en Lithops. Además, se identifica la limitación de no poder utilizar directamente Lithops en el código de Nextflow debido a la diferencia de lenguajes (Groovy y Python). Esta limitación impulsa la decisión de utilizar Flask como servidor de aplicaciones web para recibir las solicitudes de ejecución de tareas desde el executor de Nextflow y facilitar la integración con Lithops.

3.1.2. ***Diseño***

Durante esta fase, se define la arquitectura del sistema y se establecen las especificaciones técnicas. Se diseña una arquitectura basada en Flask y Gunicorn para permitir una comunicación eficiente entre Nextflow y Lithops. Se establece la estructura del servidor Flask, incluyendo la URL de escucha ("localhost:8000/process") y el formato JSON para las solicitudes de ejecución de tareas. Asimismo, se establece la lógica del servidor Flask para invocar la función lambda correspondiente a través de Lithops y ejecutar la tarea con los parámetros proporcionados. Gunicorn se integra como servidor HTTP para ejecutar múltiples instancias de Flask, garantizando así la escalabilidad y el rendimiento del sistema.

También se diseña un diagrama de secuencias que será usado para la implementación:

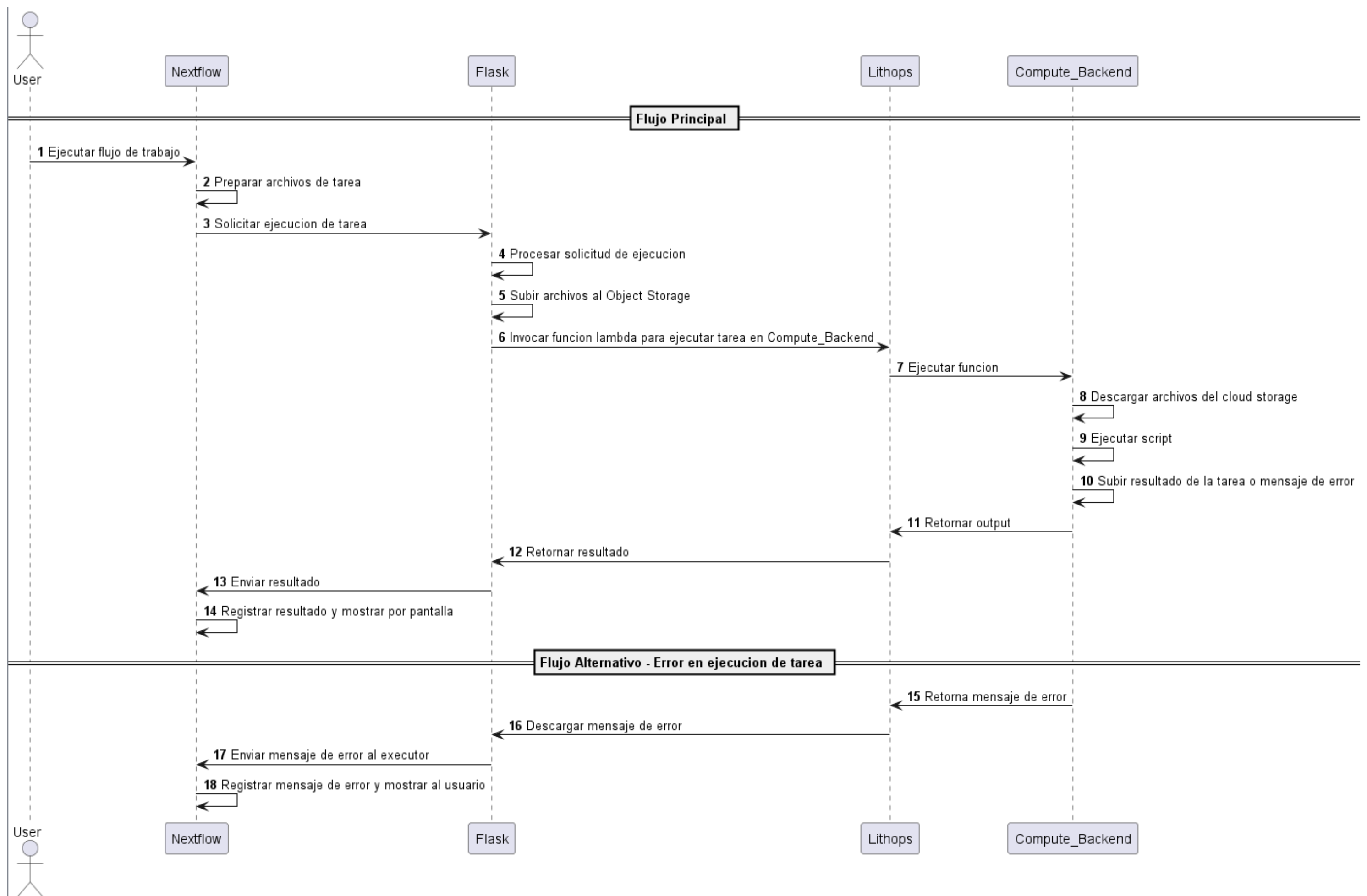

Figura 9. Diagrama de secuencias

3.1.3. Implementación

Durante esta fase, se traducen las especificaciones de diseño en código real. Se modifica la plantilla de Nextflow para que en lugar de ejecutar el archivo ".command.sh", se ejecute el comando "curl" hacia el servidor Flask con la ruta del script de la tarea. Se desarrolla el servidor Flask, implementando la lógica necesaria para recibir y procesar las solicitudes de ejecución de tareas, extraer los datos del cuerpo JSON y llamar a la función correspondiente a través de Lithops. Además, se configura Gunicorn para ejecutar múltiples instancias de Flask, asegurando una distribución equitativa de las solicitudes y mejorando la capacidad de respuesta del sistema. También se adapta el código para identificar el tipo de tarea en Nextflow (ejecución de script o ejecución de contenedor Docker) y enviar los parámetros correspondientes al servidor Flask. Se realizan pruebas

exhaustivas de integración para verificar el correcto funcionamiento del plugin de Nextflow con Lithops, asegurando una ejecución fluida de las tareas en el entorno elegido.

*3.1.4. **Pruebas***

Durante esta fase, se realizan pruebas exhaustivas para garantizar el correcto funcionamiento del sistema. Se llevan a cabo pruebas unitarias del servidor Flask para verificar su funcionalidad y asegurar una comunicación adecuada con Lithops. El código para las individuales de carga y descarga de archivos pueden encontrarse en el Anexo. También se ejecutan pruebas de rendimiento para evaluar el comportamiento del sistema bajo diferentes cargas de trabajo y garantizar tiempos de respuesta aceptables. Asimismo, se realizan pruebas de integración para verificar la correcta ejecución de tareas utilizando Lithops a través del plugin de Nextflow.

3.2.Implementación

A continuación, se muestra el diagrama del prototipo y se detallarán los pasos de la ejecución en profundidad:

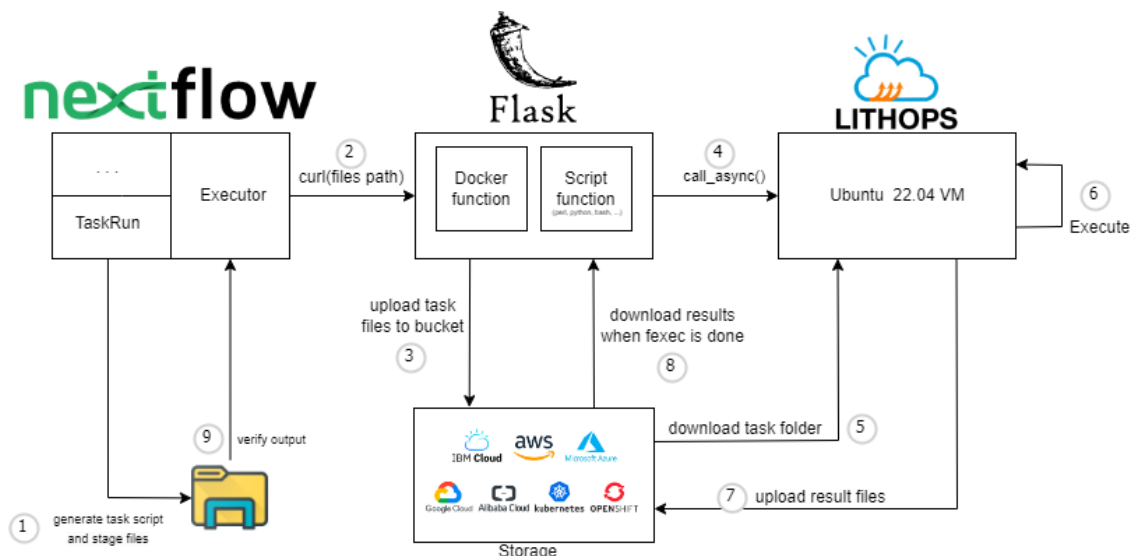

Figura 10. Diagrama ejecución LithFlow

3.2.1. Generación ficheros y petición

Como se ha mencionado previamente, Nextflow genera archivos específicos para cada tarea en el directorio "work". Cada tarea tiene una identificación única, y la estructura de carpetas y archivos suele verse de la siguiente manera:

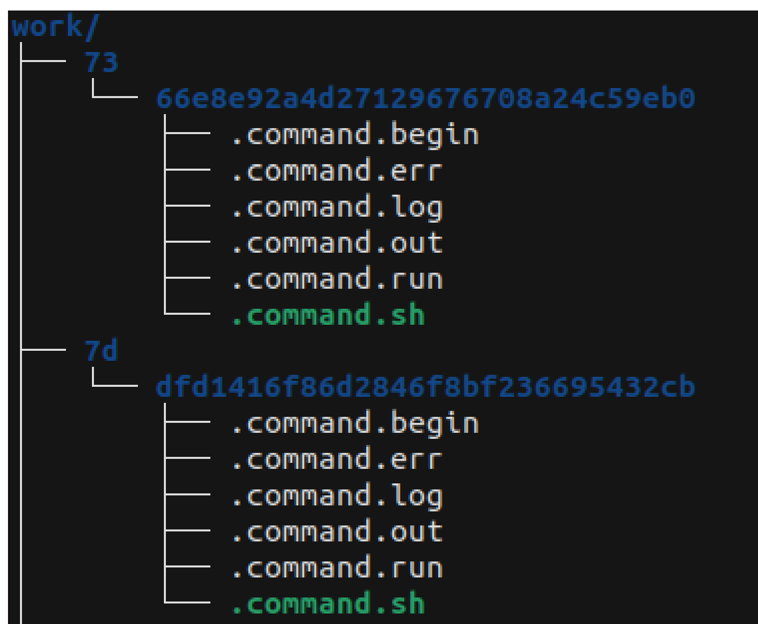

Figura 11. Ejemplo estructura ficheros work

En esta estructura, los dos caracteres aleatorios forman una subcarpeta dentro del directorio "work", y dentro de esa subcarpeta se encuentra la carpeta con la identificación única de la tarea. Dentro de cada carpeta de tarea, se encuentran los archivos relevantes para la ejecución de la tarea:

- .command.run: Este archivo contiene la lógica en forma de script Bash para ejecutar la tarea.
- .command.sh: Este archivo contiene el código de la tarea que se ejecutará.
- .command.begin: Este archivo registra el inicio de la tarea.
- .command.out: Este archivo contiene el output (salida) de la tarea.
- .command.err: Este archivo contiene el stderr (error estándar) de la tarea.
- .command.log: Este archivo registra información y registros de la ejecución de la tarea.

Estos archivos son utilizados por Nextflow para ejecutar y monitorear el progreso de las tareas, y también para capturar cualquier output o error producido durante la ejecución.

Este directorio "work" se encuentra en el directorio desde el cual se ejecuta el comando Nextflow. También se puede especificar su ubicación en cada ejecución utilizando el parámetro "--work-dir" o configurando la variable de entorno "NXF_WORK".

Dentro de la plantilla utilizada para generar el archivo .command.run, se encuentra una función de Nextflow que se ve de la siguiente manera:

```
1  nxf_launch() {
2      {{launch_cmd}}
3  }
```

Código 3. Código base de .command.run

Cuando el objeto TaskRun en el código de Nextflow crea el archivo .command.run para una tarea específica, se realiza una sustitución de {{launch_cmd}} con el comando de ejecución correspondiente. En el caso de que la tarea requiera ejecutar un script, se sustituirá por la ejecución del script en sí. En cambio, si la tarea requiere una llamada a Docker, se sustituirá por la llamada correspondiente al entorno Docker.

Sin embargo, en el prototipo desarrollado, esta lógica se ha modificado para que realice una petición al servidor Flask en función del tipo de comando. En lugar de reemplazar {{launch_cmd}} directamente con la ejecución del script o la llamada a Docker, se preparan los parámetros para la llamada al servidor Flask que enviará el comando a ejecutar y los input y output paths. Esto permite que el servidor Flask maneje la lógica de ejecución de comandos y coordine la ejecución en función del backend configurado.

De esta manera, el archivo .command.run generado contendrá la llamada al servidor Flask para enviar el comando correspondiente y esperar la respuesta.

Así es como se ve esta modificación:

```
1  nxf_launch() {
2      string='{{launch_cmd}}'
3      if [[ $string =~ ^docker ]]; then
4          # Extraer lo que va después de "--name $NXF_BOXID"
5          container=$(echo "$string" | sed -n -e 's/^.*--name $NXF_BOXID
6  //p' | awk '{print $1}')
```

```
7           # Construir el comando deseado
8           vars="eval $(nxf_container_env)"
9           command="docker run -i -v
10  /tmp/work:/home/imfx/lithflow/nxf/work/.nextflow/work $container
11  /bin/bash -c \"${vars}; cd ${output_path}; chmod +x
12  ${output_path}/.command.sh; ${output_path}/.command.sh\""
13          docker_command=$(echo $command | sed 's/"/\\"/g')
14          work_path=$(realpath "$output_path/../..")
15          # Construir la cadena JSON con los parámetros
16          json_data="{\"docker\": \"$docker_command\", \"output_path\":
17  \"$output_path\", \"work_path\": \"$work_path\"}"
18          # Hacer la solicitud curl con los parámetros JSON
19          curl -X POST -H "Content-Type: application/json" -d
20  "$json_data" http://localhost:8000/process
21      else
22          command="{{launch_cmd}}"
23          result=$(echo "$command" | cut -d' ' -f3)
24          sed -i '2i\cd /home/ubuntu' $result
25          # Construir la cadena JSON con los parámetros
26          json_data="{\"command\": \"$result\", \"input_path\":
27  \"$input_dir\", \"output_path\": \"$output_path\"}"
28          # Hacer la solicitud curl con los parámetros JSON
29          curl -X POST -H "Content-Type: application/json" -d
30  "$json_data" http://localhost:8000/process
31      fi
32  }
```

Código 4. nxf_launch() modificado

Si el comando comienza con la palabra "docker", se trata de una tarea que requiere una ejecución en un contenedor Docker. En este caso, se extrae el nombre del contenedor de la cadena de comandos y se construye el comando deseado para ejecutar el contenedor Docker. Además, se establecen las variables de entorno necesarias para la ejecución dentro del contenedor. El comando construido se ejecuta mediante la herramienta docker run, donde se especifica el volumen de montaje del directorio de trabajo (/tmp/work) y se ejecuta el archivo .command.sh dentro del contenedor.

Por otro lado, si el comando no es de tipo Docker, se considera que es un script o un comando de ejecución directa. En este caso, se extrae el comando de la cadena y se obtiene el resultado de la ejecución. Luego, se modifica el resultado para agregar una línea que cambia al directorio /home/ubuntu. Este paso es específico para el entorno de ejecución en el que se probó el prototipo.

Una vez que se tiene el comando final y el resultado de la ejecución, se construye una cadena JSON que contiene los parámetros necesarios para la solicitud al servidor Flask. En el caso de las tareas Docker, se incluye el comando Docker completo y la ruta de salida. Para las otras tareas, se incluye el resultado del comando y las rutas de entrada y salida.

Finalmente, se realiza una solicitud curl al servidor Flask en http://localhost:8000/process, pasando los parámetros JSON. Esto envía la solicitud al servidor Flask para que inicie la ejecución de la tarea correspondiente en el backend configurado.

Ciertamente, en otra parte del archivo se encuentra una función llamada nxf_stage() encargada de preparar los archivos de entrada para una tarea. Si una tarea necesita acceder a archivos generados por tareas anteriores, esta función crea enlaces simbólicos (symbolic links) en el directorio de la tarea actual que apuntan a esos archivos. De esta manera, la tarea puede acceder a los archivos necesarios sin tener que conocer la ubicación real de los mismos.

La creación de enlaces simbólicos se realiza utilizando el comando ln -s. Este comando crea un enlace simbólico en el directorio de la tarea actual que apunta al archivo en cuestión. De esta forma, cuando la tarea intenta acceder al archivo a través de la ruta del enlace simbólico, se redirige automáticamente al archivo real ubicado en el directorio de la tarea anterior.

Esta técnica de enlaces simbólicos es especialmente útil cuando las tareas dependen de archivos generados en etapas anteriores del flujo de trabajo. Permite que las tareas posteriores accedan a los datos necesarios sin tener que preocuparse por la ubicación exacta de los archivos generados previamente.

En los pasos finales de la ejecución se explicará cómo se manejan los archivos de salida y cómo se utilizan como entradas para las tareas posteriores.

3.2.2. *Lógica de Flask*

Cuando uno de los servidores Flask recibe una solicitud, se ejecuta el archivo "flask_app.py". Este archivo contiene un conjunto de funciones y lógica para procesar la solicitud y proporcionar una respuesta. Aquí solo se muestra la función principal que recibe la llamada. Sin embargo, el código completo se encuentra en el Anexo.

```
1   @app.route('/process', methods=['POST'])
2   def process():
3       print("\n=====NEW TASK RECEIVED=====\n")
4       data = request.json
5       fexec = lithops.FunctionExecutor()
6       output_path = data['output_path']
7       upload_files_to_s3("nf-s3bucket", output_path, output_path[1:])
8       if 'docker' in data:
9           docker_comm = data['docker']
10          work_path = data['work_path']
11          print("Docker command: " + docker_comm)
12          print("Output Path: " + output_path)
13          print("Work Path: " + work_path)
14          args = [docker_comm, output_path, work_path]
15          fexec.call_async(docker_func, args)
16      else:
17          command = data['command']
18          input_path = data['input_path']
19          output_path = data['output_path']
20          print("Script path: " + command)
21          print("Input Path: " + input_path)
22          print("Output Path: " + output_path)
23          args = [command, input_path, output_path]
24          fexec.call_async(script_func, args)
25      result = fexec.get_result()
26      print("\nGETTING EXECUTION OUTPUT:\n")
27      print(result)
28      print("\nDownloading directory from Storage...\n")
```

```
29        download_s3_directory("nf-s3bucket", output_path, output_path)
30        print("\nDONE\n")
31        return result
32
33    if __name__ == '__main__':
34        app.run()
35
```

Código 5. Porción del fichero "flask_app.py"

A continuación, se detalla el funcionamiento paso a paso de este archivo:

1. Importamos los módulos necesarios, como Flask, request, lithops, subprocess y Storage. Estos módulos nos permiten crear la aplicación web, manejar las solicitudes HTTP, interactuar con el backend de Lithops, ejecutar comandos en el sistema operativo y gestionar el almacenamiento en la nube.
2. Creamos una instancia de la clase Flask y la asignamos a la variable "app". Esta instancia representa nuestra aplicación web Flask.
3. Definimos varias funciones que serán invocadas en diferentes momentos de la aplicación. Estas funciones se encargan de realizar operaciones como descargar o subir archivos desde el almacenamiento en la nube, ejecutar el comando Docker o el script.
4. Configuramos una ruta llamada "/process" que escucha las solicitudes POST. Cuando se recibe una solicitud en esta ruta, se ejecuta la función "process()".
5. La función "process()" recibe la solicitud JSON y realiza las siguientes acciones:
    1. Imprime un mensaje indicando que se ha recibido una nueva tarea.
    2. Crea una instancia de la clase FunctionExecutor de Lithops para interactuar con el backend de ejecución.
    3. Obtiene el output path de los datos recibidos en la solicitud.
    4. Sube el directorio de la tarea al almacenamiento en la nube de lithops.
    5. Si los datos contienen un comando Docker, se ejecuta la función "docker_func()" de forma asíncrona con los argumentos correspondientes.

6. Si los datos contienen un comando de script, se ejecuta la función "script_func()" de forma asíncrona con los argumentos correspondientes.
7. Obtiene el resultado real de la ejecución utilizando la función "get_result()" de la instancia de FunctionExecutor.
8. Imprime el resultado de la ejecución.
9. Descarga el directorio de la tarea desde el almacenamiento en la nube.
10. Retorna el resultado.

Después de que la ejecución de la tarea haya finalizado, Nextflow realizará una verificación para asegurarse de que los archivos generados por la tarea coincidan con los archivos esperados. Esto se hace para garantizar la integridad de los resultados y asegurarse de que todo está en orden antes de continuar con la ejecución.

En cuanto al código de "flask_app.py", se han implementado optimizaciones para evitar descargar nuevamente un archivo si no ha sido modificado o para evitar subir un archivo a la nube si ya se encuentra allí. Estas optimizaciones se realizan con el objetivo de ahorrar tiempo y recursos.

El código verifica si un archivo ya existe localmente antes de descargarlo desde el almacenamiento en la nube. Si el archivo ya está presente, se evita una descarga innecesaria. Del mismo modo, antes de subir archivos a la nube, se comprueba si ya se encuentran allí para evitar una transferencia redundante.

Estas optimizaciones ayudan a reducir el tiempo de ejecución y a minimizar el uso de recursos, ya que se evitan operaciones de descarga y carga innecesarias cuando los archivos no han cambiado desde la ejecución anterior.

3.2.3. Ejecución en Lithops

Ahora se explicará el proceso realizado dentro del backend de computación de Lithops. A continuación, se detallan dos funciones: docker_func y script_func, que se ejecutarán en función del tipo de tarea que se haya enviado a Flask. A continuación, se muestra cada porción del código que se ejecuta en el backend especificado en función del tipo de tarea:

```
1   def docker_func(args_list):
2       docker_comm, output_path, work_path = args_list
```

```
3       bucket_name = 'nf-s3bucket'
4       local_directory = '/tmp/work'
5       download_work_dir(bucket_name, work_path, local_directory)
6       try:
7           docker_comm = docker_comm.replace('\\"', '"')
8           result = subprocess.check_output(['bash','-c', docker_comm],
9   stderr=subprocess.STDOUT)
10          task_dir = '/'.join(output_path.split('/')[-3:])
11          local_directory = os.path.join('/tmp/', task_dir)
12          upload_files_to_cloud(bucket_name, local_directory,
13  output_path[1:])
14      except subprocess.CalledProcessError as exc:
15          return "Status : FAIL" + str(exc.returncode) + str(exc.output)
16      except Exception as e:
17          return "\n ERROR: " + str(e) + "\n"
18      return result.decode('utf-8')
19
```

Código 6. docker_func()

La función docker_func se ejecutará cuando se reciba una tarea de tipo "docker". Recibe una lista de argumentos args_list, que incluye el comando docker a ejecutar (docker_comm), la ruta de salida (output_path) y la ruta de trabajo (work_path). Estos paths son los relativos al almacenamiento en la nube. A continuación, realiza las siguientes acciones:

1. Descarga los archivos desde el directorio del almacenamiento en la nube al directorio local /tmp/work mediante la función download_work_dir.
2. Realiza una sustitución en el comando docker para eliminar secuencias de escape innecesarias.
3. Ejecuta el comando docker utilizando subprocess.check_output. El resultado de la ejecución se almacena en la variable result.
4. Obtiene el directorio de tarea correspondiente a partir de la ruta de salida.
5. Actualiza el directorio local a /tmp/ seguido del directorio de tarea.
6. Sube los archivos generados al almacenamiento en la nube utilizando la función upload_files_to_cloud.
7. Si se produce algún error durante la ejecución del comando docker, se capturan las excepciones y se devuelve un mensaje indicando el estado de fallo.

8. Finalmente, se devuelve el resultado decodificado en formato UTF-8.

```
1   def script_func(args_list):
2       command, input_path, output_path = args_list
3       bucket_name = 'nf-s3bucket'
4       local_directory = '/home/ubuntu'
5       if input_path:
6           download_cloud_directory(bucket_name, output_path,
7   local_directory)
8       local_script_path = '/home/ubuntu/script.sh'
9       storage.download_file(bucket_name, command[1:], local_script_path)
10      if output_path:
11          existing_files = set()
12          for root, dirs, files in os.walk(local_directory):
13              for file in files:
14                  existing_files.add(os.path.join(root, file))
15      try:
16          new_mode = os.stat(local_script_path).st_mode | 0o100
17          os.chmod(local_script_path,new_mode)
18          result = subprocess.check_output(['bash', '-c',
19  local_script_path], stderr=subprocess.STDOUT)
20          if output_path:
21              upload_files_to_cloud(bucket_name, '/home/ubuntu',
22  output_path[1:])
23      except subprocess.CalledProcessError as exc:
24          return "Status : FAIL" + str(exc.returncode) + str(exc.output)
25      except Exception as e:
26          return e
27      return result.decode('utf-8')
28
```

Código 7. script_func()

Por otro lado, la función script_func se ejecutará cuando se reciba una tarea de tipo "script". También recibe una lista de argumentos args_list, que incluye el comando del script a ejecutar (command), la ruta de entrada (input_path) y la ruta de salida (output_path). A continuación, realiza las siguientes acciones:

1. Descarga los archivos desde el directorio en la nube al directorio local /home/ubuntu si hay un input path especificado, utilizando la función download_cloud_directory.
2. Descarga el script desde el almacenamiento en la nube y lo guarda en la ruta local /home/ubuntu/script.sh utilizando storage.download_file de lithops.
3. Si hay un output path especificado, se crea un conjunto de archivos existentes en el directorio local. Se usará más adelante para subir solo los ficheros modificados y nuevos tras la ejecución.
4. Cambia los permisos del script descargado para que sea ejecutable.
5. Ejecuta el script utilizando subprocess.check_output. El resultado de la ejecución se almacena en la variable result.
6. Si hay un output path especificado, se suben los archivos generados al almacenamiento en la nube utilizando la función upload_files_to_cloud.
7. Si se produce algún error durante la ejecución del script, se capturan las excepciones y se devuelve un mensaje indicando el estado de fallo.
8. Finalmente, se devuelve el resultado decodificado en formato UTF-8.

3.3.Entorno de ejecución

El proyecto se encuentra alojado en el repositorio de GitHub (https://github.com/IMFX0/lithflow). Dentro del repositorio hay dos carpetas: "nxf" y "lithopsListener". La carpeta "nxf" contiene el código de Nextflow modificado, mientras que la carpeta "lithopsListener" alberga la lógica del servidor Flask, junto con otros scripts para probar funciones específicas, como la carga y descarga de archivos o directorios.

Para ejecutar el proyecto, se siguen los siguientes pasos:

1. Descarga el repositorio desde GitHub (https://github.com/IMFX0/lithflow).
2. Abre una terminal y ejecuta el script "start_lithserver.sh", pasando como único argumento la cantidad de workers deseados. Es decir “./start_lithserver.sh 4” como ejemplo.
3. En otra terminal, navega a la carpeta "nxf" y compila el código ejecutando el comando "make compile" o, en su defecto, "./gradlew compile exportClasspath".

4. Ejecuta el script "add_env_nxf.sh" para añadir variables de entorno al archivo .bashrc. Esto asegurará que los archivos temporales, de caché y el directorio de trabajo se creen dentro de "nxf/work".
5. Una vez compilado y con las variables de entorno configuradas, ejecuta "./launch.sh run <PIPELINE>" en la carpeta "nxf". Reemplaza "<PIPELINE>" con el archivo ".nf" correspondiente o la ruta a un repositorio de nf-core junto con sus parámetros.
6. Se proporciona una pipeline de prueba en la ubicación "nxf/lithflow_tests/params_test/test.nf", la cual lee secuencias de un archivo FASTA y las invierte. Para ejecutar este test, utiliza el comando "./launch.sh run lithflow_tests/params_test/test.nf". Desde la terminal del servidor Flask podrás observar cómo se reciben y procesan las solicitudes.

4. Validación

En esta sección, se presentan los resultados obtenidos a partir de las pruebas realizadas para comparar la ejecución de una pipeline utilizando dos plataformas diferentes: AWS Batch y Lithops con Nextflow. El objetivo de estas pruebas fue evaluar el rendimiento, los costos, la facilidad de uso y realizar un análisis comparativo de los resultados obtenidos en cada plataforma.

Cabe destacar que Nextflow ofrece la capacidad de generar ficheros con los resultados de las ejecuciones. Estos ficheros son "*execution_report*" y "*execution_timeline*". Estos ficheros contienen información detallada sobre el tiempo de ejecución de cada tarea y los recursos asociados utilizados durante la ejecución. Esta información es invaluable para comprender el desempeño de la pipeline y realizar un análisis de los resultados obtenidos en cada plataforma.

4.1.Tiempos de ejecución

En primer lugar, se realizaron mediciones de los tiempos de ejecución de la pipeline en ambas plataformas. Para ello, se observaron los tiempos de inicio y finalización de cada tarea dentro de los ficheros obtenidos. Estos resultados permiten evaluar la eficiencia y el rendimiento de cada plataforma en términos del tiempo requerido para completar la pipeline. Se analizará si existe una diferencia significativa en los tiempos de ejecución y si alguna de las plataformas muestra una ventaja en términos de velocidad.

A continuación, se presenta una comparación visual del lapso de ejecución de una pipeline utilizando AWS Batch como executor. En el diagrama, las barras en gris representan el tiempo de aprovisionamiento de recursos entre otras actividades post y pre-tarea que realiza AWS, mientras que las barras de color indican el tiempo real de ejecución de cada tarea:

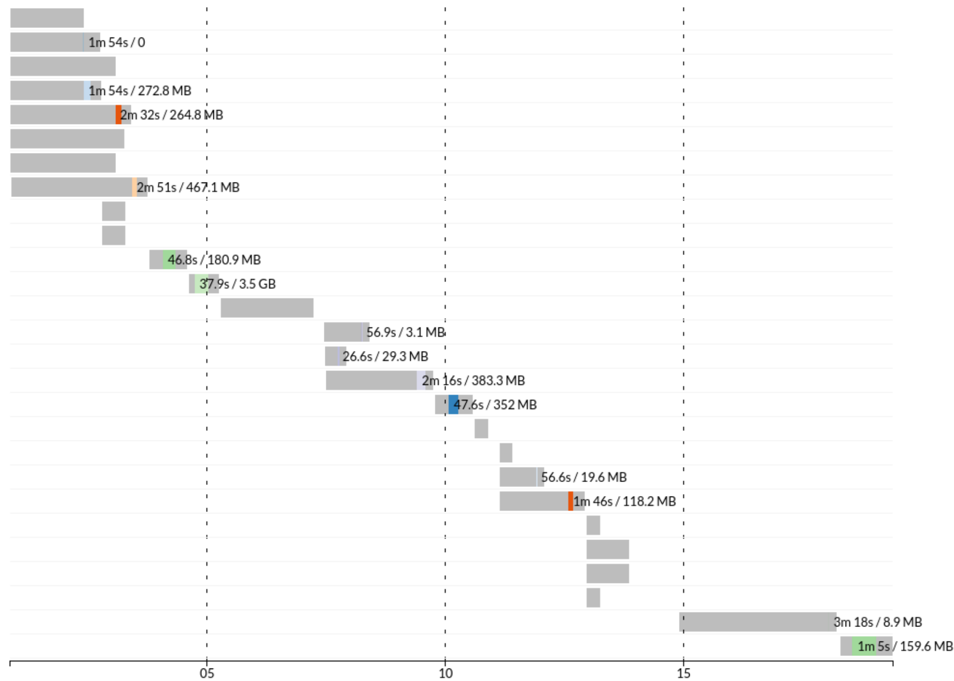

Figura 12. Execution timelapse AWS

El tiempo de ejecución de AWS Batch fue de **19m 17s**.

Al analizar visualmente el diagrama de ejecución de la pipeline utilizando AWS Batch como executor, es evidente a simple vista que la mayor parte del tiempo de ejecución no se dedica a la ejecución real de las tareas. En lugar de eso, se puede observar que una parte considerable del tiempo se destina a actividades como la preparación de las máquinas EC2 y otras tareas de aprovisionamiento de recursos.

Esta observación se hace evidente al notar que las barras en gris, que representan el tiempo de aprovisionamiento de recursos, ocupan una proporción significativa del diagrama. Mientras que las barras de color, que indican el tiempo real de ejecución de las tareas, son relativamente más cortas en comparación.

Por otro lado, se muestra el lapso de ejecución de la misma pipeline utilizando Lithops como executor:

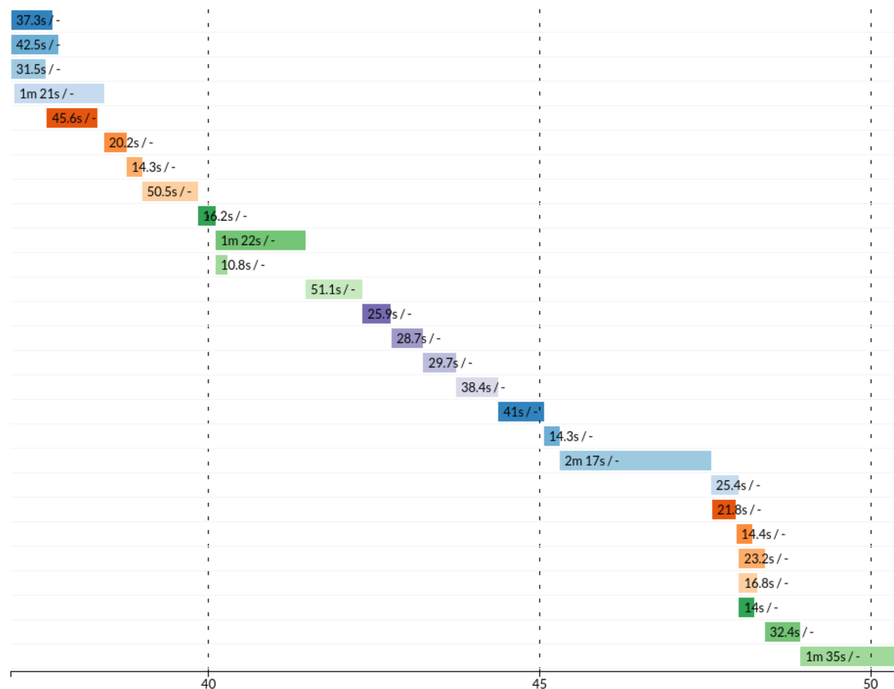

Figura 13. Execution timelapse Lithops

El tiempo de ejecución de Lithops fue de **13m 58s.**

En contraste, al analizar el diagrama de ejecución de la misma pipeline utilizando Lithops como executor, se puede apreciar que no existe una fase de aprovisionamiento de recursos como en el caso de AWS Batch. En cambio, el enfoque serverless de Lithops permite una ejecución más inmediata de las tareas, sin necesidad de esperar por la configuración de máquinas EC2 u otras actividades de aprovisionamiento.

Esta diferencia en la distribución del tiempo de ejecución entre AWS Batch y Lithops resalta la ventaja de Lithops en términos de eficiencia y aprovechamiento del tiempo de ejecución. Al evitar la fase de aprovisionamiento de recursos y centrarse directamente en la ejecución de las tareas, Lithops logra un mayor rendimiento y una reducción significativa en el tiempo total de ejecución.

En conclusión, al observar los colores representados en el diagrama, se hace evidente que en el caso de AWS Batch, la mayor parte del tiempo de ejecución se destina a actividades de aprovisionamiento de recursos, mientras que con Lithops se logra una ejecución más eficiente al evitar dicha fase y enfocarse directamente en la ejecución de las tareas.

4.2.Costes de ejecución

Nextflow, en su implementación estándar, no proporciona una funcionalidad directa para mostrar los costes de ejecución de una pipeline. Sin embargo, se ha realizado una aproximación al cálculo de costes basada en las máquinas utilizadas por cada tarea durante la ejecución.

La aproximación se basa en considerar el costo asociado al uso de la máquina virtual utilizada por cada tarea. Se ha tenido en cuenta el tipo de máquina y su precio por hora para calcular el costo estimado de ejecución de cada tarea. Es importante tener en cuenta que esta estimación se enfoca únicamente en el costo de uso de la máquina y no incluye otros posibles costos asociados, como el almacenamiento de datos o el tráfico de red. En la aproximación de coste realizada, se obtuvo un valor estimado de **0.055333$**, teniendo en cuenta el uso de las máquinas m4.xlarge y r4.large. Estas máquinas cuentan con 4 y 2 vCPU respectivamente, y una capacidad de memoria de 16 GB cada una.

Es importante destacar que al revisar los execution reports generados durante las ejecuciones, se observó que la memoria allocated utilizada por las tareas era de tan solo 6 GB y, en algunos casos, se utilizaban como máximo 2 vCPU. Esto implica que se estaban utilizando recursos de más en comparación con los recursos asignados a las tareas, lo cual podría resultar ineficiente en términos de costos.

Este hallazgo sugiere que existe una oportunidad para optimizar aún más los costos de ejecución al ajustar las configuraciones de recursos asignados a las tareas, utilizando máquinas con capacidades más acordes a los requerimientos reales de cada tarea. Esto permitiría evitar el desperdicio de recursos y, en consecuencia, reducir los costos asociados.

Por otro lado, Lithops, al ofrecer una amplia gama de opciones de backend, incluyendo la posibilidad de ejecución serverless, brinda una mayor flexibilidad en términos de costos. Esto se debe a que se pueden elegir diferentes servicios y configuraciones en

función de las necesidades de la tarea y del presupuesto disponible. Esto permite optimizar los costos y adaptarlos de manera más precisa a los requerimientos específicos de cada tarea.

Además, es importante destacar que en comparación con la ejecución en AWS Batch, el uso de Lithops resultó significativamente más económico. En los casos de ejecución anteriores, donde se utilizó una única instancia de máquina m5.large, el costo estimado fue de **0.0208$**. En contraste al costo estimado para la ejecución en AWS Batch que era de **0.055333$**, prácticamente el doble de caro.

Esta diferencia en los costos resalta la ventaja económica que ofrece Lithops en comparación con AWS Batch. La flexibilidad de Lithops al permitir utilizar diversos backends, incluyendo opciones serverless, brinda una mayor libertad para ajustar los recursos y adaptarlos a las necesidades específicas de cada tarea, lo cual se traduce en una mayor eficiencia en términos de costos.

Es importante tener en cuenta que los casos de ejecución analizados hasta ahora se refieren a pipelines de tamaño moderado. Sin embargo, a medida que se aborden pipelines más exigentes y de mayor duración, la diferencia en los costos entre Lithops y AWS Batch será aún más notable. Lithops ofrece la posibilidad de utilizar recursos de manera más eficiente y adaptarlos dinámicamente según las necesidades de cada tarea, lo que permite optimizar los costos en escenarios de mayor complejidad.

En conclusión, a pesar de que Nextflow no proporciona una funcionalidad directa para mostrar los costos de ejecución, se ha realizado una aproximación basada en el uso de máquinas virtuales. Mientras tanto, Lithops ofrece una mayor flexibilidad en términos de costos debido a su variedad de backends y opciones de ejecución, permitiendo una optimización más precisa de los costos. Los casos de ejecución anteriores demostraron que Lithops pudo lograr un costo total estimado más bajo en comparación con el uso de AWS Batch.

4.3.Facilidad de uso

La facilidad de uso y configuración es un aspecto clave a considerar al comparar Lithops con AWS Batch en el contexto de Nextflow. Mientras que Lithops ofrece una experiencia más ágil y flexible, AWS Batch puede resultar engorroso y requerir una serie de pasos adicionales.

En el caso de Lithops, la configuración del entorno es bastante sencilla. Solo se necesita instalar Lithops como una biblioteca de Python y proporcionar las credenciales adecuadas para el proveedor de servicios en la nube que se elija, como AWS, Azure o Google Cloud. Una vez configurado, Lithops permite seleccionar fácilmente el backend deseado, como máquinas virtuales, contenedores o incluso entornos serverless. Esto brinda una gran flexibilidad y adaptabilidad al poder ajustar el entorno de ejecución según las necesidades específicas del proyecto.

Por otro lado, la configuración de AWS Batch con Nextflow puede resultar más compleja y engorrosa. Para utilizar AWS Batch, se requiere la creación de una AMI (Amazon Machine Image) personalizada que contenga todos los requisitos de software y dependencias necesarios para la ejecución de la pipeline. Esto implica tiempo y esfuerzo adicional para preparar y mantener dicha AMI. Además, es necesario tener los roles y permisos adecuados configurados en AWS para permitir el acceso y la interacción con los servicios necesarios.

Además, la configuración de la infraestructura de computación y la cola de trabajo en AWS Batch también puede ser un proceso complejo. Se deben definir y configurar correctamente las instancias EC2, las políticas de escalado, las colas de trabajo y los recursos asociados. Esto puede llevar tiempo y requerir conocimientos técnicos especializados.

En contraste, Lithops simplifica este proceso al proporcionar una interfaz más intuitiva y abstracta. La capacidad de seleccionar diferentes backends con Lithops, incluyendo

opciones serverless, reduce la necesidad de configuraciones específicas y permite un enfoque más flexible en la asignación de recursos.

Además de su facilidad de uso y configuración, Lithops destaca por su transparencia y facilidad para detectar y solucionar errores en comparación con AWS Batch en Nextflow.

Lithops proporciona una mayor visibilidad y transparencia en cuanto a errores y registros de ejecución. Durante la ejecución de la pipeline, Lithops genera registros detallados que permiten rastrear y diagnosticar cualquier problema que pueda surgir. Estos registros son accesibles directamente desde la salida de la consola, lo que facilita la identificación de errores y la depuración de problemas en tiempo real. Esta transparencia contribuye a una experiencia más eficiente y eficaz al permitir una rápida detección y resolución de problemas.

En contraste, AWS Batch puede requerir más tiempo y esfuerzo para encontrar y analizar los registros y errores. Los registros de AWS Batch se almacenan en CloudWatch, lo que puede implicar una navegación adicional y una búsqueda más minuciosa para identificar la fuente de un error. Esto puede resultar en un proceso más lento y complicado para diagnosticar y solucionar problemas durante la ejecución de la pipeline.

En resumen, Lithops destaca por su facilidad de uso, configuración y transparencia en comparación con AWS Batch en el contexto de Nextflow. La sencillez en la creación del entorno Lithops y la flexibilidad para elegir diferentes backends brindan una experiencia más ágil y adaptable. Por otro lado, AWS Batch puede resultar más engorroso debido a la necesidad de configurar una AMI personalizada, establecer roles y permisos adecuados, y configurar la infraestructura de computación y la cola de trabajo. En términos de facilidad y opciones de configuración, Lithops se presenta como una solución más conveniente.

4.4.Valoración final

A la conclusión a la que se llega es que Lithops se destaca como una opción superior a los executors de Nextflow en la nube, ya sea AWS Batch, Azure Batch o cualquier otro que siga una ejecución tipo batch. Esto se debe a varias razones:

- Eficiencia de ejecución: El análisis de los diagramas muestra que Lithops logra una ejecución más eficiente al reducir la fase de aprovisionamiento de recursos, lo que resulta en tiempos de ejecución más rápidos en comparación con AWS Batch. Además, al adaptar los recursos asignados a las tareas de manera más precisa, Lithops evita el desperdicio de recursos y optimiza los costos asociados.

- Optimización de costos: Lithops ofrece una mayor flexibilidad en términos de costos al permitir la selección de diferentes backends, incluyendo opciones serverless. Esto permite ajustar los recursos y adaptarlos de manera más precisa a las necesidades específicas de cada tarea. En comparación, AWS Batch puede resultar significativamente más costoso, como se evidenció en los casos de ejecución anteriores donde Lithops demostró ser prácticamente el doble de económico.

- Facilidad de uso y configuración: Lithops se destaca por su facilidad de uso y configuración. Solo se requiere instalar Lithops como una biblioteca de Python y proporcionar las credenciales adecuadas para el proveedor de servicios en la nube elegido. Además, la posibilidad de seleccionar diferentes backends con Lithops simplifica el proceso y evita la necesidad de configuraciones específicas. En comparación, AWS Batch puede resultar engorroso al requerir la preparación de una AMI personalizada, la configuración de roles y permisos, y la definición de la infraestructura de computación y la cola de trabajo.

- Transparencia y manejo de errores: Lithops se distingue por su transparencia y facilidad para detectar y solucionar errores. Los registros detallados y accesibles directamente desde la consola permiten una identificación rápida y una resolución

eficiente de problemas en tiempo real. En contraste, AWS Batch puede requerir más tiempo y esfuerzo para encontrar y analizar los registros y errores almacenados en CloudWatch.

En general, Lithops ofrece una solución más eficiente, flexible y fácil de usar en comparación con AWS Batch como executor en Nextflow. Su capacidad para ejecutar tareas de manera rápida y económica, adaptar los recursos de manera precisa, y proporcionar una experiencia transparente y fácil de usar, hacen de Lithops una elección sólida para aquellos que buscan maximizar la eficiencia y minimizar los costos en sus pipelines.

5. Futuro del plugin

En este apartado, se explorarán las posibilidades y mejoras que podrían ser implementadas en el futuro para el proyecto. Aunque el plugin ya ofrece funcionalidades valiosas, siempre hay espacio para el crecimiento y la evolución. Es por ello que se ha realizado esta lista con posibles objetivos a futuro para el proyecto:

- Integración oficial en Nextflow: Una posible evolución del proyecto sería trabajar en la integración del plugin en el propio código fuente de Nextflow, convirtiéndolo en un componente oficialmente respaldado por el equipo de Nextflow. Esto permitiría una mayor visibilidad y adopción de la solución, brindando a los usuarios una experiencia aún más sólida y confiable.

- Mejor gestión y monitorización de recursos: El plugin podría incluir funcionalidades avanzadas para la gestión y monitorización de recursos en la nube, como la detección automática de cuellos de botella, la optimización dinámica de recursos asignados a las tareas, o la generación de métricas y estadísticas detalladas sobre el rendimiento y uso de los recursos.

- Integración con sistemas de gestión de colas externos: Se podría explorar la integración del plugin con sistemas de gestión de colas externos, como Slurm o SGE, para permitir a los usuarios aprovechar la infraestructura existente y mejorar la escalabilidad y rendimiento de sus pipelines.

- Incorporación de funciones de recuperación de errores: El plugin podría implementar mecanismos de recuperación de errores automatizados, como la reejecución de tareas fallidas o la implementación de estrategias de tolerancia a fallos, para mejorar la robustez y fiabilidad de las ejecuciones de pipelines.

- Mejoras en la documentación y comunidad de usuarios: Sería beneficioso desarrollar una documentación completa y detallada del plugin, incluyendo ejemplos de uso, guías de configuración y solución de problemas. Además, fomentar la creación de una comunidad de usuarios activa y colaborativa, donde los usuarios puedan compartir experiencias, contribuir con mejoras y proporcionar soporte entre sí.

6. Conclusiones

En este trabajo de fin de grado, se ha abordado el desarrollo y la evaluación de un plugin de ejecución en Nextflow utilizando la plataforma Lithops en entornos de computación en la nube. A lo largo del proyecto, se han cumplido los objetivos planteados inicialmente, superando los desafíos y adquiriendo conocimientos en áreas como el procesamiento de datos a gran escala, el uso de flujos de trabajo y la integración con servicios en la nube.

A pesar de comenzar con poco conocimiento en el ámbito del cloud computing y genómica, se ha logrado comprender y aplicar conceptos relacionados con el procesamiento distribuido en la nube. Esto ha implicado una comprensión de los procesos de análisis de datos en este campo.

En términos de resultados, se ha logrado implementar un plugin funcional que permite la ejecución de pipelines de Nextflow en el entorno de Lithops. Los tiempos de ejecución y los costos asociados han sido evaluados, demostrando una mejora significativa en comparación con otras soluciones, como AWS Batch. Además, se ha destacado la facilidad de uso y configuración de Lithops, así como su transparencia en la detección y solución de errores.

En conclusión, este trabajo de fin de grado ha sido un logro significativo, demostrando la capacidad de abordar desafíos técnicos y adquirir conocimientos en áreas complementarias. El desarrollo del plugin de ejecución en Nextflow utilizando Lithops ha cumplido con los objetivos planteados y ha demostrado su eficacia y potencial en términos de rendimiento y eficiencia. Este proyecto sienta las bases para futuras mejoras y contribuciones en el campo de los flujos de trabajo y el procesamiento de datos en entornos de computación en la nube.

Referencias

[1] *Nextflow*. URL: https://www.nextflow.io/

[2] *Lithops*. URL: https://lithops-cloud.github.io/

[3] *CloudLab*. URL: https://cloudlab.urv.cat/

[4] *AWS*. URL: https://aws.amazon.com/es/

[5] *Microsoft Azure*. URL: https://azure.microsoft.com/es-es/

[6] *Google Cloud*. URL: https://cloud.google.com/?hl=es

[7] *nf-core*. URL: https://nf-co.re/

[8] *FUSE Layer*. URL: https://en.wikipedia.org/wiki/Filesystem_in_Userspace

[9] *DSL*. URL: https://en.wikipedia.org/wiki/Domain-specific_language

[10] FASTA. URL: https://en.wikipedia.org/wiki/FASTA_format

[11] *FASTQ*. URL: https://en.wikipedia.org/wiki/FASTQ_format

Anexo

A Ficheros carga y descarga para pruebas individuales

A.1. lithops_uploader.py

```
1   import argparse
2   import os
3
4   from lithops import Storage
5
6   storage = Storage()
7
8   def upload_files_to_s3(bucket_name, local_directory, output_path):
9       # Obtener la lista de archivos generados después de la ejecución
10      work_dir = set()
11      for root, dirs, files in os.walk(local_directory):
12          for file in files:
13              work_dir.add(os.path.join(root, file))
14
15      # Filtrar los archivos nuevos y subirlos a S3
16      for file_path in work_dir:
17          s3_key = os.path.join(output_path, os.path.relpath(file_path,
18  local_directory))
19          storage.upload_file(file_path, bucket_name, s3_key)
20
21  def main():
22      parser = argparse.ArgumentParser(description='Upload files to
23  S3.')
24      parser.add_argument('--bucket', help='S3 bucket name')
25      parser.add_argument('--local-dir', help='Local directory path')
26      parser.add_argument('--output-path', help='Output path in S3')
27
28      args = parser.parse_args()
29
30      if not all([args.bucket, args.local_dir, args.output_path]):
```

```
31            parser.error('Missing required arguments')
32
33        upload_files_to_s3(args.bucket, args.local_dir,
34    args.output_path[1:])
35
36    if __name__ == '__main__':
37        main()
```

Código 8. lithops_uploader.py

A.2 lithops_downloader.py

```
1     from flask import Flask, request
2     import lithops
3     import subprocess
4     from lithops import Storage
5     import os
6     import argparse
7
8     storage = Storage()
9
10    def download_s3_directory(bucket_name, s3_path, local_directory):
11        prefix_dir = s3_path[1:]
12
13        # Obtener la lista de objetos en el directorio de S3
14        response = storage.list_keys(bucket_name, prefix=prefix_dir)
15
16        for s3_key in response:
17            local_file_path = os.path.join(local_directory,
18    os.path.basename(s3_key))
19
20            # Descargar el archivo desde S3
21            storage.download_file(bucket_name, s3_key, local_file_path)
22
```

```
23
24  def main():
25      parser = argparse.ArgumentParser(description='Download files from
26  lithops Storage')
27      parser.add_argument('--bucket', help='S3 bucket name')
28      parser.add_argument('--local-dir', help='Local directory path')
29      parser.add_argument('--output-path', help='Output path')
30
31      args = parser.parse_args()
32
33      if not all([args.bucket, args.local_dir, args.output_path]):
34          parser.error('Missing required arguments')
35
36      download_s3_directory(args.bucket, args.local_dir,
37  args.output_path)
38
39  if __name__ == '__main__':
40      main()
```

Código 9. lithops_downloader.py

B Servidor flask

B.1. flask_app.py

```
1   from flask import Flask, request
2   import lithops
3   import subprocess
4   from lithops import Storage
5   import os
6
7   storage = Storage()
8
9   app = Flask(__name__)
10
11  def download_s3_directory(bucket_name, s3_path, local_directory):
12      prefix_dir = s3_path[1:]
13
14      # Obtener la lista de objetos en el directorio de S3
15      response = storage.list_keys(bucket_name, prefix=prefix_dir)
16
17      for s3_key in response:
18          local_file_path = os.path.join(local_directory,
19  os.path.relpath(s3_key, prefix_dir))
20          local_file_dir = os.path.dirname(local_file_path)
21          os.makedirs(local_file_dir, exist_ok=True)
22
23          # Verificar si el archivo ya existe localmente
24          if not os.path.exists(local_file_path):
25              # Descargar el archivo desde S3 solo si no existe
26  localmente
27              storage.download_file(bucket_name, str(s3_key),
28  str(local_file_path))
29          else:
30              print(f"El archivo {local_file_path} ya existe localmente.
31  No se descargará nuevamente.")
32
33
34  def download_work_dir(bucket_name, s3_path, local_directory):
```

```
35        prefix_dir = s3_path[1:]
36
37        # Obtener la lista de objetos en el directorio de S3
38        response = storage.list_keys(bucket_name, prefix=prefix_dir)
39        for s3_key in response:
40            task_dir = s3_key[len(prefix_dir)+1:]
41            local_file_path = local_directory + '/' + task_dir
42
43            os.makedirs(os.path.dirname(local_file_path), exist_ok=True)
44
45            # Verificar si el archivo ya existe localmente
46            if not os.path.exists(local_file_path):
47                # Descargar el archivo desde S3 solo si no existe
48    localmente
49                storage.download_file(bucket_name, str(s3_key),
50    str(local_file_path))
51            else:
52                print(f"El archivo {local_file_path} ya existe localmente.
53    No se descargará nuevamente.")
54
55
56    def upload_files_to_s3(bucket_name, local_directory, output_path):
57        for root, dirs, files in os.walk(local_directory):
58            for file in files:
59                file_path = os.path.join(root, file)
60                s3_key = os.path.join(output_path,
61    os.path.relpath(file_path, local_directory))
62                storage.upload_file(file_path, bucket_name, s3_key)
63
64            for directory in dirs:
65                directory_path = os.path.join(root, directory)
66                sub_output_path = os.path.join(output_path,
67    os.path.relpath(directory_path, local_directory))
68                upload_files_to_s3(bucket_name, directory_path,
69    sub_output_path)
70
71
72    def docker_func(args_list):
73        docker_comm, output_path, work_path = args_list
```

```
74
75      bucket_name = 'nf-s3bucket'
76
77      # Descargar los archivos desde el directorio de S3 a /tmp/work
78      local_directory = '/tmp/work'
79      download_work_dir(bucket_name, work_path, local_directory)
80
81      # Ejecutar el docker
82      try:
83          docker_comm = docker_comm.replace('\\"', '"')
84          result = subprocess.check_output(['bash','-c', docker_comm],
85  stderr=subprocess.STDOUT)
86
87          task_dir = '/'.join(output_path.split('/')[-3:])
88
89          local_directory=os.path.join('/tmp/', task_dir)
90
91          upload_files_to_s3(bucket_name, local_directory,
92  output_path[1:])  # Subir los archivos generados a S3
93
94      except subprocess.CalledProcessError as exc:
95          return "Status : FAIL" + str(exc.returncode) + str(exc.output)
96      except Exception as e:
97          return "\n ERROR: " + str(e) + "\n"
98
99      return result.decode('utf-8')
100
101
102 def my_function(args_list):
103
104     command, input_path, output_path = args_list
105     bucket_name = 'nf-s3bucket'
106     local_directory = '/home/ubuntu'
107
108     if input_path:
109         # Descargar los archivos desde el directorio de S3 a /tmp/
```

```
110         download_s3_directory(bucket_name, output_path,
111 local_directory)
112
113     # Ejecutar el script descargado
114     local_script_path = '/home/ubuntu/script.sh'
115     storage.download_file(bucket_name, command[1:], local_script_path)
116
117     if output_path:
118         # Obtener la lista de archivos existentes en el directorio de
119 salida
120         existing_files = set()
121         for root, dirs, files in os.walk(local_directory):
122             for file in files:
123                 existing_files.add(os.path.join(root, file))
124
125     try:
126         new_mode = os.stat(local_script_path).st_mode | 0o100
127         os.chmod(local_script_path,new_mode)
128         result = subprocess.check_output(['bash', '-c',
129 local_script_path], stderr=subprocess.STDOUT)
130         if output_path:
131             upload_files_to_s3(bucket_name, '/home/ubuntu',
132 output_path[1:])  # Subir los archivos generados a S3
133
134     except subprocess.CalledProcessError as exc:
135         return "Status : FAIL" + str(exc.returncode) + str(exc.output)
136     except Exception as e:
137         return e
138
139     return result.decode('utf-8')
140
141
142 @app.route('/process', methods=['POST'])
143 def process():
144     print("\n=====NEW TASK RECIEVED=====\n")
145     data = request.json
146     fexec = lithops.FunctionExecutor()
147     output_path = data['output_path']
```

```
148
149         print("\nUploading directory to Storage...\n")
150
151         upload_files_to_s3("nf-s3bucket", output_path, output_path[1:])
152
153         if 'docker' in data:
154             docker_comm = data['docker']
155
156             work_path = data['work_path']
157             print("Docker command: " + docker_comm)
158             print("Output Path: " + output_path)
159             print("Work Path: " + work_path)
160             args = [docker_comm, output_path, work_path]
161             fexec.call_async(docker_func, args)
162
163         else:
164             command = data['command']
165             input_path = data['input_path']
166             output_path = data['output_path']
167             print("Script path: " + command)
168             print("Input Path: " + input_path)
169             print("Output Path: " + output_path)
170             args = [command, input_path, output_path]
171             fexec.call_async(my_function, args)
172
173         result = fexec.get_result()  # Obtener el resultado real del
174     ResponseFuture
175         print("\nGETTING EXECUTION OUTPUT:\n")
176         print(result)
177
178         print("\nDownloading directory from Storage...\n")
179         download_s3_directory("nf-s3bucket", output_path, output_path)
180         print("\nDONE\n")
181
182         return result
183
184     if __name__ == '__main__':
```

```
185        app.run()
```

Código 10. flask_app.py

UNIVERSITAT ROVIRA i VIRGILI