ARQUITECTURA DISTRIBUIDA PARA EL ANÁLISIS DE TORMENTAS SOLARES

DISTRIBUTED ARCHITECTURE FOR SOLAR STORMS ANALYSIS

Trabajo de Fin de Grado
Curso 2024-2025

UNIVERSIDAD
COMPLUTENSE
MADRID

Autor:
Víctor Giménez Chillada
Directores:
José Luis Vázquez Poletti
Juan Antonio Clemente Barreira
Grado en Ingeniería del Software
Facultad de Informática

NOTA FINAL: 8

ÍNDICE DE CONTENIDOS

ÍNDICE DE FIGURAS

CAPÍTULO 1. RESUMEN

Los riesgos que pueden suponer las tormentas solares para la civilización moderna podrían provocar grandes desastres. Es por esto que el objetivo de este trabajo de fin de grado (TFG) se centra en la detección de las erupciones solares, que son las desencadenantes de las tormentas solares.

Para esta detección se ha decidido utilizar tecnologías de procesamiento de imágenes que permiten distinguir las erupciones en imágenes del Sol que han sido extraídas de una misma fuente fiable, la cual asegura la uniformidad, el Solar Dynamics Observatory (SDO)[1] .

La lógica del algoritmo implementado está escrita en Python debido a su versatilidad. Además, para realizar el procesamiento de dichas imágenes, se ha desarrollado una arquitectura sin servidor (“serverless”) utilizando herramientas de Amazon Web Services (AWS) como AWS Lambda y AWS Simple Storage Service (S3), que emplean una arquitectura distribuida. El uso de una arquitectura sin servidor se debe a la importancia del procesamiento en tiempo real y el rápido tiempo de respuesta que se necesita.

Finalmente se obtuvo un sistema en el que al almacenar imágenes en un contenedor de AWS S3, estas son procesadas por un algoritmo utilizando AWS Lambda. En este algoritmo se detecta el disco solar para extraer su radio, se ocultan partes de la imagen que no resultan útiles para la detección, se transforma la imagen a binario (la imagen pasa a tener únicamente píxeles blancos o negros) y se le aplican dos filtros distintos para detectar las erupciones solares. En caso de que se detecte una erupción, las imágenes son devueltas junto con algunos metadatos (datos relacionados con la imagen obtenidos durante el procesamiento) a otro contenedor de S3.

Palabras Clave: erupción solar, arquitectura distribuida, arquitectura serverless, procesamiento de imágenes, Python, AWS S3, AWS Lambda, bucket.

[1] https://sdo.gsfc.nasa.gov/

CHAPTER 1. ABSTRACT

The risks that solar storms can imply to modern civilization could lead to major disasters. This is why the focus of this bachelor of sciences (BSc) thesis is on the detection of solar flares, which are the triggers of solar storms.

For this detection, it has been decided to use image processing technologies to distinguish flares in images of the sun that have been extracted from the same reliable source, which ensures uniformity, the Solar Dynamics Observatory (SDO)[2].

The implemented algorithm logic is written in Python due to its versatility. Furthermore, a serverless architecture has been developed to process these images, using Amazon Web Services (AWS) tools such as AWS Lambda and AWS Simple Storage Service (S3), which employ a distributed architecture. The serverless architecture use is encouraged by the importance of real time processing and the fast response time required.

Finally I obtained a system where images stored in an AWS S3 container are processed by an algorithm using AWS Lambda. In this algorithm the solar disc is detected to extract its radius, the parts of the image that are not useful for the detection are hidden, the image is transformed to binary (the image pixels turn into white or black) and two different filters are applied to detect solar flares. In case a flare is detected, the images and some metadata (image related data obtained during the processing) are returned to another S3 container.

Key Words: solar flare, distributed architecture, serverless architecture, image processing, Python, AWS S3, AWS Lambda, bucket.

[2] https://sdo.gsfc.nasa.gov/

CAPÍTULO 2. INTRODUCCIÓN

En esta sección se resumirá el motivo detrás de la creación del proyecto junto con el contexto del mismo, además de las metas alcanzadas y el plan de trabajo seguido.

2.1. Antecedentes

Las erupciones solares se pueden definir como explosiones que se producen en la superficie del Sol debido a la reconexión magnética, un proceso en el que las líneas del campo magnético del Sol se reconfiguran de forma explosiva[3]. Estas se pueden distinguir debido a la intensa luz que emiten, que puede durar desde unas pocas horas hasta unos cuantos días.

La energía liberada por una erupción solar se puede comparar con la de miles de bombas nucleares o millones de erupciones volcánicas. Emiten una gran cantidad de radiación electromagnética, tanto rayos X, rayos gamma, radiación ultravioleta y ondas de radio. En muchos casos además estas pueden ir acompañadas de eyecciones de masa coronal también conocidas como “coronal mass ejection” (CMEs) que liberan partículas cargadas en forma de plasma[4].

Las erupciones solares siguen un sistema de clasificación con letras y números basado en el estándar establecido por la National Oceanic and Atmospheric Administration (NOAA), que está reconocido por la comunidad internacional y es utilizado por instituciones como la NASA y la Agencia Espacial Europea (European Space Agency o ESA). Las clases en orden ascendente por la intensidad de los rayos X que emiten son A, B, C, M y X. Además, dentro de cada clase, existe una subclasificación con números del 1 al 10 cuyo valor indica la intensidad relativa dentro de la misma. Las únicas erupciones que exceden este rango de números son las clase X, que han llegado a sobrepasar el 10 una decena de veces en los últimos 25 años[5].

La detección de estas erupciones es importante debido a las terribles consecuencias que pueden llegar a tener si no se está preparado para ellas. Las erupciones solares, por un lado, pueden afectar a la capa exterior de la atmósfera terrestre y las comunicaciones por radio, y las CMEs, por otro lado, son capaces de dañar equipos electrónicos y poner en riesgo a los astronautas que se encuentren fuera de la protección del campo magnético terrestre. Además, son capaces de

[3] https://www.eltiempo.es/noticias/meteopedia/erupcion-solar
[4] https://www.esa.int/Science_Exploration/Space_Science/What_are_solar_flares
[5] https://www.spaceweatherlive.com/en/solar-activity/top-50-solar-flares.html

provocar sobrecargas en las redes eléctricas y dañar satélites mediante lo que se conoce como tormentas geomagnéticas, que son el resultado de la interacción de las partículas y radiación liberadas en la erupción con el campo magnético terrestre [1].

2.2. Objetivos

El objetivo del proyecto es la creación de un sistema que permita la detección de estas erupciones solares a partir de imágenes del Sol de forma automatizada con una arquitectura distribuida.

Para conseguir esto será necesario:

- Desarrollar un algoritmo en Python que permita la identificación de erupciones solares a partir de imágenes.
- Implementar un sistema en la nube usando AWS para automatizar el flujo de procesamiento.
- Almacenar los resultados procesados y los datos generados de forma estructurada.

2.3. Plan de trabajo

Para la realización del proyecto se ha dividido la planificación en una serie de fases que se detallarán a continuación:

1. Primero se debe encontrar la fuente de información de la que se obtendrán las imágenes a ser procesadas, que, a nivel de formato, deben ser lo más uniformes posible.

2. A continuación se debe elegir la plataforma para llevar a cabo una arquitectura distribuida teniendo en cuenta las necesidades del proyecto, tanto el almacenamiento de las imágenes como el procesamiento de las mismas.

3. Acto seguido, se seleccionará el lenguaje de programación más adecuado para la manipulación de las imágenes. Se deberá elegir uno que sea compatible con la plataforma anteriormente mencionada y que disponga de

un mayor número de librerías para aumentar la versatilidad en la codificación y la facilidad de su mantenimiento.

4. Después, se lleva a cabo la creación del algoritmo de procesamiento con el lenguaje seleccionado en el paso anterior y adaptado a las necesidades de la plataforma seleccionada también anteriormente.

5. Finalmente se realiza la configuración del entorno en la plataforma (tanto la política de permisos como los roles de ejecución), para posteriormente relacionar los contenedores destinados al almacenamiento de imágenes con la función de procesamiento creada.

6. Por último, tiene lugar la validación final del algoritmo realizado y del flujo de procesamiento creado con imágenes que lleven al límite su capacidad.

CHAPTER 2. INTRODUCTION

This section will summarize the rationale behind the creation of the project and its context, as well as the goals set and the work plan followed.

2.1. Background

Solar flares can be defined as explosions that occur on the surface of the sun due to magnetic reconnection, a process in which the Sun's magnetic field lines are explosively reconfigured[6]. They can be distinguished by the intense light they produce, which can last from a few hours to a few days.

The energy released by a solar flare can be compared to thousands of nuclear bombs or millions of volcanic eruptions. They emit a large amount of electromagnetic radiation, including X rays, gamma rays, ultraviolet radiation and radio waves. In many cases these can also be accompanied by coronal mass ejections (CMEs) that release charged particles in the form of plasma[7].

Solar flares have a classification system with letters and numbers based on the standard established by the National Oceanic and Atmospheric Administration (NOAA), which is recognised by the international community and used by institutions such as NASA and the European Space Agency (ESA). The classes, in ascending order by the intensity of the X rays they emit are A, B, C, M and X. Furthermore, within each class, there is a subclassification with numbers from 1 to 10 whose value is associated with the relative intensity inside the class. The only ones that exceed this range of numbers are the X class flares, which have surpassed 10 around ten times in the last 25 years[8].

The detection of these flares is important because of the terrible consequences that can take place if you are not prepared for them. Solar flares, on the one hand, can affect the outer layer of the Earth's atmosphere and radio communications, and CMEs, on the other hand, are capable of damaging electronic equipment and endanger astronauts outside the protection of the Earth's magnetic field. They are also capable of overloading power grids and damaging satellites through a phenomenon called geomagnetic storms [1].

---

[6] https://www.eltiempo.es/noticias/meteopedia/erupcion-solar
[7] https://www.esa.int/Science_Exploration/Space_Science/What_are_solar_flares
[8] https://www.spaceweatherlive.com/en/solar-activity/top-50-solar-flares.html

2.2. Objectives

The objective of the project is to create a system that allows the detection of these solar flares from images of the sun in an automated way with a distributed architecture.

To achieve this, it will be necessary to:

- Develop an algorithm in Python that allows the identification of solar flares from images.
- Implement a cloud system using AWS to automate the processing flow.
- Store the processed results and generated data in a structured way.

2.3. Work schedule

In order to carry out the project, the planning has been divided into a series of phases, which will be detailed below:

1. First, we must find the information source from which the images to be processed will be obtained, and they must be as uniform as possible at format level.

2. Then, it is important to choose the platform to implement a distributed architecture keeping in mind the requirements of the project, both image storage and image processing.

3. Afterwards, we will also select the most suitable programming language for image manipulation. One that is compatible with the above mentioned platform and provides a larger number of libraries to increase its versatility.

4. Subsequently starts the creation of the processing algorithm with the language selected in the previous step and adapted to the previously selected platform requirements.

5. Finally comes the platform environment configuration (both the permissions policy and the execution roles), in order to relate the image storage containers with the processing function created.

6. Lastly, the algorithm and processing flow final validation takes place with images that push its capacity to the limit.

CAPÍTULO 3. ESTADO DEL ARTE

A continuación se detallarán posibles arquitecturas de procesamiento en la nube a utilizar, algunos “frameworks” de procesamiento de imágenes que podrían ser útiles y trabajos similares que tocan temas relevantes para el proyecto.

3.1. Arquitecturas de procesamiento en la nube

Para la realización del proyecto se utilizó una arquitectura distribuida[9], debido principalmente a su capacidad para integrar distintas tecnologías y a la capacidad de este tipo de arquitecturas de procesar un gran volumen de imágenes. Este tipo de arquitectura se basa en la utilización de un conjunto de equipos o sistemas independientes (nodos) para que actúen como uno solo. De esta forma se consigue descentralizar el procesamiento y mejorar el rendimiento del sistema en muchos casos a un coste más bajo.

Por otro lado, la arquitectura de procesamiento en la nube permite utilizar recursos alojados en servidores remotos a través de Internet. Además, el uso de una arquitectura distribuida es esencial para el procesamiento en la nube, ya que permite la escalabilidad del sistema (añadiendo más nodos cuando sea necesario). También ofrece una gran tolerancia a fallos gracias a la posibilidad de utilizar otros nodos en caso de que alguno falle. Y cabe destacar la posibilidad de paralelizar el trabajo debido al uso de múltiples nodos simultáneamente, lo cual reduce el tiempo de ejecución de las tareas.

.

Los tipos de servicio típicamente desplegados en la nube[10] se detallarán a continuación.

3.1.1. Infraestructura como servicio (“Infrastructure as a Service” o IaaS)

La IaaS[11], como su propio nombre indica, ofrece una infraestructura como servicio, ya sean servidores, almacenamiento, redes o virtualizaciones a través de Internet.

9 https://ilimit.com/blog/importancia-arquitectura-distribuida/#que-es-un-sistema-distribuido-y-sus-caracteristicas

10 https://cloud.google.com/discover/types-of-cloud-computing

11 https://cloud.google.com/learn/what-is-iaas?hl=es

De este modo, el usuario no necesita configurar ni gestionar los recursos utilizados, ya que de esto se encarga el proveedor en la nube. Un ejemplo de IaaS es Amazon Elastic Compute Cloud (EC2)[12], que permite ejecutar máquinas virtuales personalizables en la nube.

3.1.2. Plataforma como servicio ("Platform as a Service" o PaaS)

Por otro lado, PaaS[13] permite ofrecer una plataforma con herramientas y entornos configurados por el proveedor en la nube para desarrollar aplicaciones sin necesidad de preocuparse por las actualizaciones del sistema operativo y las herramientas de desarrollo, además de no tener que mantener el hardware. Es por esto por lo que este entorno puede ser ideal para desarrolladores que simplemente quieran centrarse en el código y no quieran gestionar la infraestructura que hay detrás. Google App Engine[14] por ejemplo, permite crear aplicaciones sin ninguna de estas preocupaciones.

3.1.3. Software como servicio ("Software as a Service" o SaaS)

Al igual que los tipos anteriormente mencionados, el modelo SaaS[15] ofrece un servicio, en este caso lo que se ofrece es un software específico para que los usuarios puedan acceder por medio de un navegador a través de Internet. Además, dado que el proveedor en la nube es el encargado de gestionar todo, no es necesario ningún tipo de mantenimiento ni instalación por parte de los usuarios. Algunos ejemplos de software ofrecidos bajo demanda son, Dropbox[16], para el almacenamiento en la nube; Microsoft 365[17] que da acceso a herramientas como Word, Outlook o Excel; y Overleaf[18] para la edición de texto en LaTeX.

3.1.4. Serverless computing

En la computación sin servidor el proveedor de la nube asigna dinámicamente los recursos necesarios para la ejecución. Por lo tanto, el usuario no necesita

[12] https://aws.amazon.com/es/ec2/
[13] https://cloud.google.com/learn/what-is-paas?hl=es
[14] https://cloud.google.com/appengine?hl=es-419
[15] https://cloud.google.com/saas?hl=es
[16] https://www.dropbox.com/es_ES/
[17] https://www.office.com/
[18] https://es.overleaf.com/

encargarse de gestionar la infraestructura del servidor para ejecutar las funciones o tareas que componen la aplicación. Esto permite reducir el coste y mejorar la eficiencia, por lo que su uso es ideal para aplicaciones basadas en eventos.

A diferencia de otros tipos de servicio en los que el usuario debe gestionar los servidores virtuales (IaaS), donde se ofrecen entornos completos de desarrollo (PaaS) o se brinda acceso a un software completo (SaaS), la computación sin servidor se centra en funciones más pequeñas que se activan por eventos. Un ejemplo de esto es AWS Lambda, que permite ejecutar funciones Lambda, que son bloques de código que se activan a través de desencadenadores como la subida de archivos en un “bucket”, que es un contenedor de almacenamiento de objetos, lo cual resulta muy útil para el proyecto.

3.2. Proyectos relacionados

Antes de empezar con el proyecto era necesario explorar otros trabajos relacionados para asegurarse de no desarrollar algo que ya existe. En esta búsqueda se encontraron 2 proyectos muy relacionados y cuya información ha sido vital para el desarrollo de este proyecto, pero también se investigó otro trabajo que finalmente resultó no ser tan relevante [2].

3.2.1. A serverless computing architecture for Martian aurora detection with the Emirates Mars Mission

Para explicar el objetivo del proyecto es necesario entender el contexto con respecto a las auroras en Marte y la misión “Hope” o “Emirates Mars Mission” (EMM).

Existen dos tipos de auroras marcianas:

1. Auroras discretas: pueden aparecer en cualquier lugar donde los campos magnéticos de la corteza sean más débiles, lo cual permite el acceso de los electrones a la atmósfera. Esto se debe a la falta de un campo magnético en el planeta, que provoca que las auroras no se focalicen en las regiones polares como en la Tierra.

2. Auroras difusas: estas son causadas por la precipitación de electrones y protones de alta energía. Suelen producirse en el lado diurno de Marte y están más dispersas y con límites peor definidos que las discretas.

Por otro lado, “Hope” es la nave espacial que orbita alrededor de Marte, operada por el Centro Espacial Mohammed bin Rashid (Emiratos Árabes Unidos), y su objetivo es proporcionar una imagen completa de la atmósfera marciana. La órbita que sigue es única y le permite observar todas las regiones geográficas y detectar cualquier fenómeno que ocurra en el planeta. Además cuenta con un espectrómetro ultravioleta que le permite observar rangos de longitud de onda específicos que lo convierten en un instrumento ideal para la detección de las auroras.

Gracias a estas tecnologías se han detectado nuevos tipos de auroras discretas con distintas propiedades, sin embargo aún no se ha comprendido la física detrás de estas. Es esto lo que motivó la creación de un proyecto capaz de detectar y dar información de estas nuevas auroras.

El proyecto recibe la financiación de la European Union´s Horizon 2020 bajo el acuerdo Marie Skłodowska-Curie[19]. Y cuenta con la colaboración de la New York University Abu Dhabi (NYUAD) junto con la Universidad Complutense de Madrid, que han trabajado con datos proveídos por el EMM Science Data Center (SDC)[20], que es un repositorio donde se guardan todos los datos científicos de la misión.

Con este proyecto se busca aportar un nuevo enfoque a la EMM. Se utiliza una arquitectura sin servidor para analizar las auroras marcianas, las cuales son muy importantes para entender la atmósfera de Marte. Se utilizan bibliotecas como OpenCV y algoritmos de “machine learning” para realizar clasificación de imágenes, detección de objetos y segmentaciones de forma efectiva. Todo esto acompañado de la capacidad de manejar grandes cantidades de imágenes gracias a la computación en la nube, permite resolver problemas complejos y abrir el camino para futuros proyectos del campo de la detección remota.

También es importante destacar el uso de este proyecto como referencia para la creación de otros trabajos relacionados. Por ejemplo, un estudio acerca de las auroras marcianas en el que se intenta entender el motivo por el que estas aparecen en ciertos momentos o lugares específicos, encontrando también relaciones entre las auroras y la geografía del planeta, el ángulo cenital del sol, la hora local o las estaciones [3]. Otro trabajo que ha utilizado este proyecto como referencia trata sobre la conversión automática de diagramas dibujados a mano en formatos digitales, permitiendo generar topologías de red usando técnicas de machine learning a través de modelos de YOLO. Aunque el tema no está relacionado con el proyecto de esta sección, se utilizó como referencia debido al uso

[19] https://cordis.europa.eu/project/id/101007638/es
[20] https://sdc.emiratesmarsmission.ae/

de SageMaker para las detecciones de las figuras, evitando así la dependencia de los recursos del ordenador [4].

El uso de este proyecto como base para la realización del trabajo de fin de grado se basa precisamente en la similitud de los procesos a realizar, ya que en ambos proyectos se intenta llevar a cabo una detección remota sobre imágenes y se utiliza una arquitectura sin servidor.

El algoritmo y procedimiento de detección utilizado en el proyecto de las auroras marcianas sirve de gran inspiración para la detección de las erupciones solares, principalmente la transformación de las imágenes. En el proyecto de Marte se aplican máscaras y transformaciones de la imagen a binario para facilitar la detección de las auroras, método que será replicado para detectar las erupciones solares en este proyecto. No se entrará en detalle ahora sobre este procedimiento ya que será explicado más adelante.

Por otro lado la arquitectura sin servidor utilizada en este proyecto se construye sobre otras herramientas como AWS S3, utilizado como servicio de almacenamiento, y AWS Lambda, para la computación sin servidor. Por lo tanto, se trasladará el uso de estos servicios a este TFG, ya que se prueba que pueden resultar muy útiles para guardar las imágenes y generar eventos que las procesen mediante un algoritmo [5].

3.2.2. Machine learning model survey with the dataset for solar flare prediction

Este proyecto es una investigación acerca de la predicción de las erupciones solares realizada para el Student Research Symposium de la Embry-Riddle Aeronautical University (ERAU)[21] de 2021. Aunque no recibió ningún tipo de financiación de esta institución. A diferencia del anterior proyecto, en este hay un único autor perteneciente a PeopleTec Inc. Sin embargo, recibe asesoramiento de otras personas gracias al programa PeopleTec, Inc. Technical Fellows[22] y al Dr. Jonathan W. Campbell de la ERAU.

Dado que este proyecto también se centra en las erupciones solares, el contexto es similar al de este trabajo de fin de grado. Sin embargo, también se mencionan erupciones solares catastróficas ocurridas en el pasado.

[21] https://commons.erau.edu/db-srs/2021/thesis-session/4/
[22] https://www.peopletec.com/about-peopletec/technical-fellows/

Por un lado, el evento Carrington en 1859, que se cree que fue la tormenta solar más fuerte de los últimos 500 años, y se considera la más fuerte registrada en la historia. Aunque no se disponía de la tecnología para medir la intensidad de la misma, se registraron numerosas auroras boreales y hubo interrupciones en las redes de telégrafo utilizadas en esa época. Por lo tanto, si un evento así se produjese hoy en día, se cree que los daños tendrían un coste de trillones de dólares.

Por otro lado, la New York Railroad Storm de 1924 se calcula que tuvo una intensidad similar al evento Carrington, pero no se puede saber con certeza debido a la falta de tecnología del momento en ambos eventos. En este caso, antes de las perturbaciones del campo magnético se detectó una gran actividad por parte de los espectroheliógrafos en zonas del Sol que presentaban grupos de manchas solares. Esta actividad de manchas solares previas a la erupción llevó a la idea de que las erupciones solares podrían llegar a predecirse.

La finalidad del proyecto es similar a la de este TFG, sin embargo, busca predecir las erupciones solares en lugar de detectarlas. Para esta predicción utiliza datos extraídos de satélites que le permiten reaccionar con un mayor tiempo de antelación y anticiparse a las erupciones. El procedimiento utilizado para llevar a cabo las predicciones emplea distintos modelos de “machine learning” y los compara para comprobar la eficiencia de cada uno. Así mismo, verifica también qué parámetros de las tablas de las que se extraen los datos son más relevantes para predecir una erupción. Además, la motivación detrás de la predicción es más extrema, ya que no surge a raíz de las erupciones solares habituales sino de las más catastróficas mencionadas anteriormente [6].

Por otro lado, este TFG tiene como objetivo detectar las erupciones a partir de imágenes tomadas con un telescopio de la NASA (el SDO) en vez de utilizar valores extraídos de tablas. Por lo tanto su finalidad no es la misma y no es necesario el uso de machine learning.

CAPÍTULO 4. ARQUITECTURA DESARROLLADA

La arquitectura elegida finalmente se centra en un enfoque “serverless”, utilizando los servicios de AWS debido a las grandes ventajas que ofrece, como por ejemplo, su facilidad de uso, flexibilidad, rentabilidad, escalabilidad, desempeño y seguridad. En concreto se utilizan los servicios de AWS S3 y AWS Lambda, pero dentro de estos también se emplean un gran número de herramientas y tecnologías diferentes.

4.1. Herramientas y tecnologías utilizadas

En este apartado se especificarán todas y cada una de las herramientas y tecnologías elegidas finalmente debido a sus capacidades y compatibilidad con los requisitos del sistema. Además, se detallará el papel que estas cumplen dentro de este trabajo de fin de grado.

4.1.1. Amazon Web Services

Amazon Web Services se puede definir como un proveedor de servicios y plataformas en la nube utilizado a nivel mundial. Los servicios utilizados en este trabajo de fin de grado son resumidos en esta sección, pero se explicarán más en detalle en las secciones 4.2 y 4.3.

Por un lado, se utiliza Amazon Simple Storage Service, también conocido como Amazon S3[23] que es un servicio de almacenamiento de objetos que ofrece escalabilidad, disponibilidad de datos, seguridad y rendimiento. Esta herramienta permite optimizar costes, organizar los datos y configurar controles de acceso.

Por otro lado, AWS Lambda[24] es un servicio que sigue un modelo de computación en la nube sin servidor (ya descrito anteriormente en la sección 3.1). Este ajusta automáticamente los recursos necesarios según la demanda, permite el uso de múltiples lenguajes y es ideal para realizar tareas basadas en eventos. Su principal función es la ejecución del código en una infraestructura de alta disponibilidad a través de las funciones Lambda. Por lo tanto, para utilizarlo sólo es necesario

[23] https://docs.aws.amazon.com/es_es/AmazonS3/latest/userguide/Welcome.html
[24] https://docs.aws.amazon.com/es_es/lambda/latest/dg/welcome.html

suministrar el código en uno de los lenguajes compatibles con Lambda, en este caso Python.

Una vez se organice el código en funciones de Lambda, estas se ejecutarán sólo cuando sea necesario mediante desencadenadores. Solo se cobran los costes del tiempo informático que se consuma, lo que hace que sea una herramienta muy económicamente accesible para todo tipo de usuarios.

Uno de los aspectos más destacables de utilizar estas herramientas es la facilidad de integración entre ambas, permitiendo a Lambda ejecutarse en base a eventos como subidas de archivos en S3, y facilitando la interacción con bibliotecas como boto3, para descargar, subir o crear archivos en "buckets" desde la propia función Lambda.

4.1.2. Python

El uso del lenguaje de programación Python[25] se caracteriza por su amplia disponibilidad de bibliotecas especializadas, versatilidad y simplicidad a la hora de programar.

Python cuenta con un gran número de bibliotecas especializadas para el procesamiento de imágenes, se dispone de Pillow, NumPy o OpenCV, y para trabajar en la nube con AWS Lambda existe boto3 que permite la interacción con otros servicios de AWS, por ejemplo, AWS S3.

Por otro lado, Python tiene una sintaxis sencilla y fácil de entender que reduce la complejidad del código para su uso en AWS Lambda, lo cual resulta muy útil para el procesamiento de imágenes, ya que en principio la transformación de una imagen según unos parámetros establecidos podría resultar algo complicado.

Finalmente, otro de los puntos a favor de Python es su popularidad, ya que al ser muy conocido y utilizado, existen muchos recursos, ejemplos y soporte para trabajar con AWS y las bibliotecas que sean necesarias, y en conclusión, hay mucha documentación.

[25] https://aws.amazon.com/es/what-is/python/

4.1.3. Visual Studio Code

Visual Studio Code (o VS Code), es un editor de código fuente gratuito, ligero, rápido y personalizable que dispone de extensiones y herramientas que otorgan flexibilidad al usuario.

Sus principales características son “el autocompletado inteligente mediante IntelliSense, la depuración integrada, y el control de versiones con Git, lo que convierte a VS Code en una opción versátil tanto para desarrolladores individuales como para equipos”[26].

Uno de los principales motivos por los que se ha elegido este editor es la experiencia del autor con el mismo. Sin embargo, también resultaba especialmente útil dado que lo que se desarrolló en un inicio antes de la integración con AWS era un script en Python, por lo que gracias a la capacidad de VS Code de establecer puntos de interrupción, comúnmente conocidos como “breakpoints”, y la ejecución de código paso a paso, permitía una depuración mucho más rápida del código. Además, VS Code cuenta con una terminal integrada para ejecutar scripts y administrar dependencias, lo cual aumenta aún más la rapidez del desarrollo y la depuración.

4.1.4. Bibliotecas

En esta sección se resumirán las bibliotecas utilizadas en el proyecto, tanto las necesarias para el procesamiento de imágenes como otras secundarias cuyo uso también ha sido muy importante. Cabe decir que elegir un “framework” de procesamiento de imágenes correcto es una de las decisiones más importantes para el proyecto, ya que es la funcionalidad principal del mismo.

4.1.4.1. OpenCV

OpenCV[27], abreviatura de Open Source Computer Vision Library, es una librería de código abierto que proporciona funciones en Python para procesamiento de imágenes mediante algoritmos de visión por computador. Esta biblioteca contiene funcionalidades que sirven para la detección de objetos, el reconocimiento facial, el procesamiento de vídeo, etc… Se suele utilizar para tareas de análisis de vídeos en tiempo real o sistemas de detección de patrones.

[26] https://openwebinars.net/blog/que-es-visual-studio-code-y-que-ventajas-ofrece/
[27] https://imaginaformacion.com/tutoriales/opencv-en-python

En este caso, para la detección del disco solar en las imágenes y la obtención de su radio, se han utilizado funciones de OpenCV que permiten cargar imágenes, convertir estas a escala de grises, aplicar un desenfoque y detectar círculos. En la sección 4.4 se entrará más en detalle sobre qué funciones se han utilizado exactamente y se explicarán a un nivel más técnico.

4.1.4.2. Pillow

Pillow[28] es una biblioteca que permite abrir imágenes, manipularlas y guardarlas en distintos formatos. La idea principal de esta biblioteca es el acceso rápido a datos guardados en formatos de píxeles básicos, lo que significa que se puede acceder a los valores numéricos que representan los colores de cada píxel y manipularlos.

Ha sido la biblioteca más utilizada durante el proyecto y sus principales casos de uso han sido:

1. La apertura de imágenes para su manipulación.

2. La creación de imágenes y el dibujo de figuras sobre estas, creando así máscaras que se superponen sobre las imágenes de las erupciones, lo cual permite ocultar partes no deseadas de las mismas.

3. La conversión de imágenes a binario y escala de grises.

4.1.4.3. NumPy

Aunque NumPy[29] no es específicamente una biblioteca de procesamiento de imágenes, permite manejar vectores y matrices de datos de forma rápida y eficiente, lo cual resulta muy útil para el análisis y la manipulación de imágenes. Es por esto por lo que muchas bibliotecas de procesamiento de imágenes (como OpenCV o Pillow) la utilizan internamente. Al permitir la representación de imágenes como vectores o matrices, se pueden realizar operaciones básicas sobre ellas, como cambiar el tamaño o aplicar filtros.

El uso de NumPy no es del todo relevante en el proyecto, ya que tan solo se utiliza una vez e incluso en ese caso podría ser sustituido por otras funciones. Sin

[28] https://pillow.readthedocs.io/en/latest/
[29] https://aprendeconalf.es/docencia/python/manual/numpy/

embargo, se utiliza para mantener la consistencia del proyecto, ya que el resultado devuelto por la función de detección de círculos ya mencionada en la sección 3.2.1 de OpenCV es una matriz Numpy, y la función de Numpy empleada se encarga de redondear los valores devueltos. Además, el procesamiento con esta función de redondeo que ofrece NumPy es algo más rápido que con otros métodos.

4.1.4.4. Boto3

Boto3 es una biblioteca oficial de Python que sirve para interactuar con AWS, ya que facilita la conexión y gestión de los servicios de Amazon en la nube. Gracias al uso de Boto3 se ha podido trabajar con los “buckets” creados en S3 desde la propia función Lambda.

Su uso específico ha sido la descarga de las imágenes subidas en el “bucket” origen para su posterior manipulación y la carga de las imágenes en el “bucket” destino en caso de cumplir los requisitos de detección para las erupciones.

4.1.4.5. JSON

Por último, la biblioteca JSON permite trabajar con el formato JSON (JavaScript Object Notation), que se suele utilizar para compartir y almacenar datos en conjuntos clave-valor organizados de forma estructurada y fácil de leer. Además de lo ya mencionado, se ha utilizado este formato por su compatibilidad con un gran número de lenguajes de programación, su popularidad en el mundo del desarrollo y la informática, y su simplicidad para estructurar la información.

En este proyecto, esta biblioteca solamente se ha utilizado para guardar los metadatos relacionados con las imágenes en las que se han detectado erupciones solares. Estos JSON se guardan en archivos de formato “.json” para facilitar su posterior gestión por el usuario en caso de requerirlo.

4.2. Integración de Amazon S3 en este proyecto

Para almacenar los datos en AWS S3 es necesario utilizar “buckets”, que como ya se mencionó anteriormente son contenedores para objetos almacenados en Amazon S3. La cantidad de objetos que se permiten almacenar en un mismo “bucket” es ilimitada, pero los costes aumentarán en función del peso de los objetos. Es importante mencionar que los “buckets” poseen un nombre único para

diferenciarlos, ya que se puede acceder a estos de forma global según los permisos de acceso otorgados al mismo.

En el caso de nuestro proyecto se utilizan 2 “buckets” necesariamente:

1. El primero se llama “bucket-origen-erupcion-solar” y es donde se cargarán las imágenes que deben ser procesadas para detectar las erupciones solares (ver Figura 1). Gracias a otra herramienta que se explicará en el apartado 4.3, al subir una imagen se procesará automáticamente y en caso de detectarse una erupción solar se subirá la misma imagen a nuestro segundo “bucket”.

2. El segundo “bucket”, llamado “bucket-destino-erupcion-solar” se utilizará para almacenar, tanto las imágenes en las que se detecten erupciones solares, como un archivo JSON con información relativa a la erupción solar detectada tal y como se puede observar en la Figura 2.

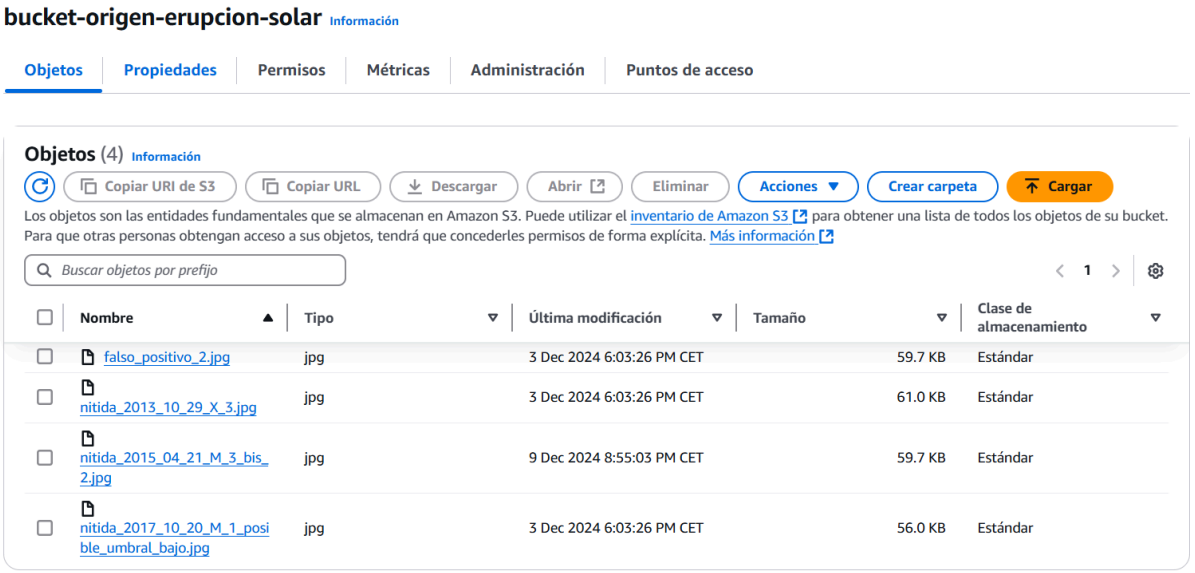

Figura 1. “Bucket” origen

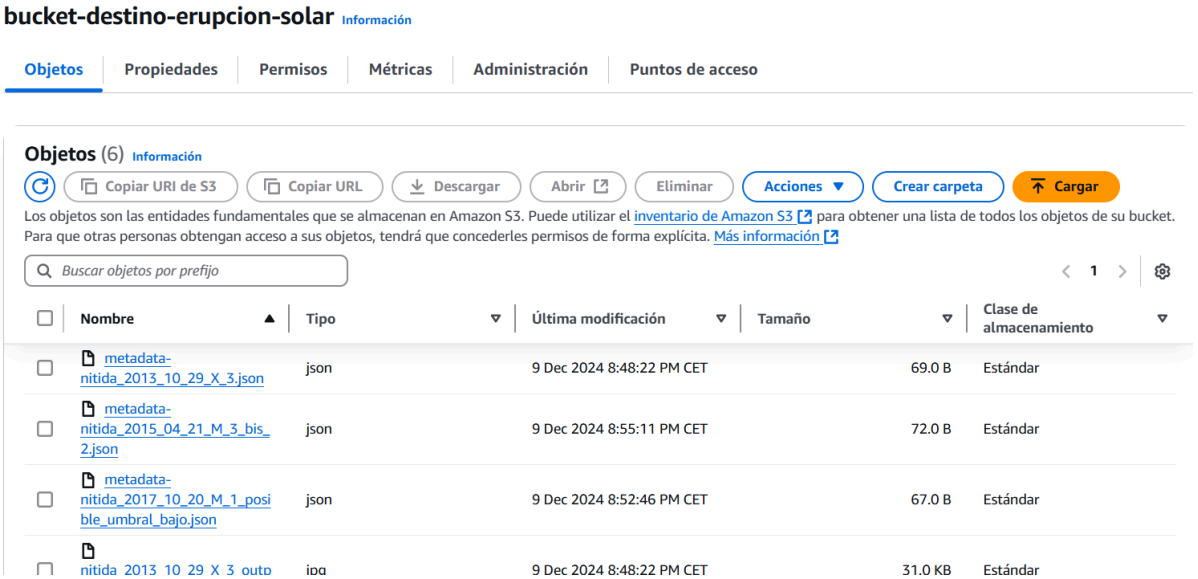

Figura 2. "Bucket" destino

4.3. Integración de AWS Lambda en este proyecto

En este trabajo de fin de grado, el código suministrado a AWS Lambda se utiliza para procesar las imágenes subidas al "bucket" de origen descrito en la sección 4.2. Para esto es necesario vincular la función a un disparador que hará que se ejecute cada vez que se suba un archivo al "bucket". Además, gracias al uso de AWS Lambda con una arquitectura distribuida se pueden procesar múltiples imágenes de forma masivamente paralela.

Tan solo es necesario el uso de una única función (DetectSolarFlare, ver Figura 3) dentro de AWS Lambda, la cual se encarga de procesar la imagen, hacer las transformaciones y comprobaciones pertinentes, y generar un archivo JSON con la información más relevante para añadirlo a dicho "bucket" destino junto con la imagen originalmente subida.

Es importante remarcar que aunque se ha hecho referencia a una única función Lambda, esta puede estar compuesta por muchas funciones más pequeñas ("`circle_detection()`", "`create_mask_image()`", etc...) que son llamadas dentro de una función principal ("`lambda_handler()`"), por lo que se puede asemejar más a un "script" que a una función como tal. Todas estas funciones se explicarán más en detalle en la sección 4.4.

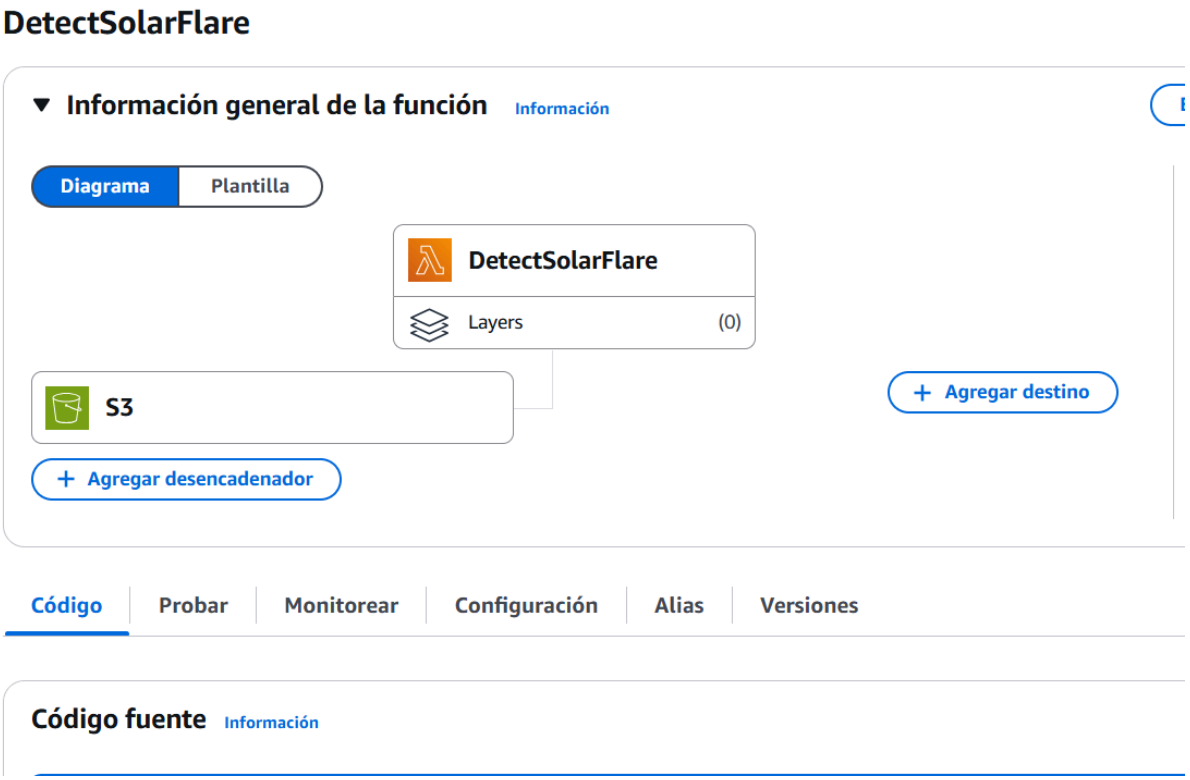

Figura 3. Función Lambda

La arquitectura general de todo el proceso está descrita gráficamente en la Figura 4.

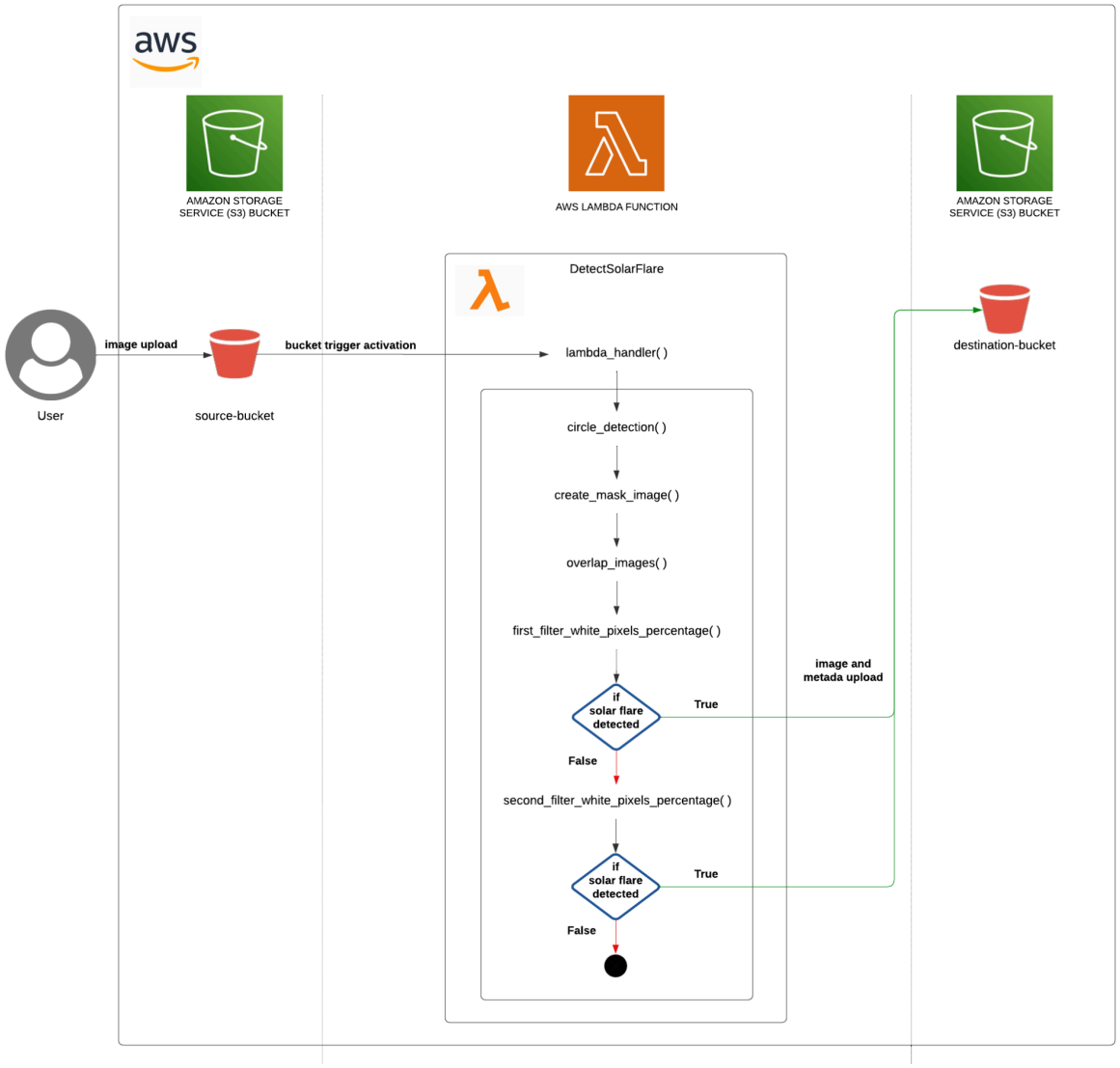

Figura 4. Arquitectura del sistema basada en la creación de 2 "buckets" de almacenamiento y una función Lambda para el procesamiento de una imagen

4.4. Desarrollo del proyecto

Una vez enumeradas las herramientas y tecnologías utilizadas en este TFG, así como descrita la arquitectura del sistema, en esta sección se explica cómo se han utilizado esas herramientas para la creación del proyecto final. Con el fin de simplificar la explicación, en esta sección sólo se repasará el algoritmo de Python para el procesamiento de imágenes. Por otro lado, la parte de configuración del entorno en la nube y sin servidor se dejará para la sección del manual de usuario y la preparación del entorno.

4.4.1 Repaso del código

Primero se hará un resumen de los pasos seguidos a lo largo del algoritmo, empezando por la función "`lambda_handler()`" que recibe como parámetro el evento que ha desencadenado la llamada a la función Lambda.

Dentro de esta función principal, primero se lleva a cabo la extracción de los nombres de "buckets", archivos y rutas.

Tras guardar estos nombres en sus respectivas variables se obtiene el radio del círculo solar con "`circle_detection()`". Este paso es necesario porque aunque todas las imágenes se hayan extraído de la misma fuente (el SDO), en función de la fecha en la que se haya tomado la foto el zoom de las imágenes puede cambiar y mostrar el Sol con pequeñas variaciones de tamaño.

A continuación, se crea una máscara para cubrir todas las partes de la imagen que no resultan útiles para la detección de erupciones con "`create_mask_image()`" y se solapa a la imagen original con "`overlap_images()`", de forma que se obtenga una sola imagen como la subida originalmente pero con ciertas partes cubiertas de una capa negra.

Antes de explicar el siguiente paso, que es la aplicación del primer filtro, se debe comprender el motivo por el cual se creó. El objetivo inicial del proyecto era la detección de las erupciones solares originadas en el borde del disco solar o cercanas a él, que son las únicas en las que se puede apreciar la expulsión de materia al exterior debido a la representación de las imágenes en 2 dimensiones. Sin embargo, se detectó que en ocasiones las imágenes tienen mucho ruido visual (distorsión de los píxeles que degrada la calidad de la imagen) cuando las fotografías son tomadas en el momento más intenso de la erupción, probablemente debido al deslumbramiento que provocan las mismas. Este ruido, que en principio puede parecer un problema, se ha utilizado para detectar erupciones que se producen en el interior del disco solar, ya que se ha verificado con un gran número de imágenes que aparece únicamente con el deslumbramiento de las erupciones solares. Por lo tanto, el objetivo del primer filtro es la detección de erupciones buscando este ruido visual en las imágenes. La función que aplica el primer filtro "`first_filter_white_pixels_percentage()`", transforma la imagen enmascarada a binario de modo que se pueda apreciar el ruido visual en forma de píxeles blancos. De esta manera, se pueden detectar los casos en los que la imagen tenga mucho ruido visual calculando el porcentaje de píxeles blancos de la misma tras aplicarle el filtro.

Una vez se hace la comprobación del primer filtro, si no se ha detectado nada se procede a aplicar el segundo filtro. Este también transforma la imagen a binario, pero lo hace de una forma mucho más estricta, de manera que sólo las partes realmente luminosas de la imagen se mantienen como manchas blancas. Esto resulta verdaderamente útil ya que las erupciones que se dan en el borde del disco solar expulsan plasma acompañado de un destello. Por lo que, tras aplicar este segundo filtro, la forma del plasma liberado se ve como una mancha blanca sobre una imagen totalmente negra. Y del mismo modo que antes, calculando el porcentaje de píxeles blancos de la imagen con este filtro y comparándolo con el valor umbral establecido (se detallará más acerca de los valores umbral en la sección 4.4.2), se puede concluir si hay una erupción solar en la imagen procesada.

Finalmente se guardan las imágenes y algunos metadatos relacionados (en formato JSON) en el "bucket" destino.

4.4.1.1. Importación de bibliotecas

Lo primero que se debe hacer es importar las bibliotecas "`boto3`", "`os`", "`sys`", "`uuid`", "`unquote_plus`" de "`urllib.parse`", "`numpy`", "`Image`" junto con "`ImageDraw`" de "`PIL`" (pillow), "`cv2`" y "`json`" al proyecto como se puede ver en la Figura 5.

```
import boto3
import os
import sys
import uuid
from urllib.parse import unquote_plus
import numpy as np
from PIL import Image, ImageDraw
import cv2
import json
```

Figura 5. Bibliotecas utilizadas

4.4.1.2. Preparación y configuración inicial

Una vez importadas las bibliotecas se debe implementar la función "`lambda_handler()`", que es donde empieza la ejecución del código y el punto de

entrada principal de la función Lambda “DetectSolarFlare” creada en AWS Lambda. Esta función tiene 2 parámetros de entrada tal y como se puede ver en la Figura 6. El parámetro “`event`”, es un diccionario que contiene los datos enviados a la función Lambda durante su invocación, en este caso serán datos relacionados con el “bucket” de S3 desde el que se activó el evento. Por otro lado, el parámetro “`context`” es un objeto que proporciona información sobre el tiempo de ejecución de la función Lambda y el entorno, pero en este proyecto no se utilizará.

```
def lambda_handler(event, context):
```

Figura 6. Función lambda_handler

Dentro de la función se itera sobre los distintos registros contenidos en el evento “`event`” (ver Figura 7). Esto se debe a que aunque el comportamiento normal de la función es que cada vez que se suba un archivo al “bucket” se genere un evento independiente (de forma que se puedan procesar muchas subidas de archivos simultáneas de forma paralela), pueden darse casos en los que las subidas de archivos simultáneas generen una llamada a la función Lambda con múltiples entradas. Con este bucle se asegura que en estos casos raros se traten todas las subidas de archivos y no solo una de ellas.

```
def lambda_handler(event, context):
  for record in event['Records']:
```

Figura 7. Bucle de eventos

A continuación se explicará el código aplicado a cada una de las imágenes subidas en el “bucket”. Primero se establece el nombre del “bucket” origen con la información obtenida de la entrada del evento, y a continuación se asigna el valor a la variable que guarda el nombre del “bucket” destino, como se puede ver en la Figura 8.

```
for record in event['Records']:
  source_bucket = record['s3']['bucket']['name']
  destination_bucket = 'bucket-destino-erupcion-solar'
```

Figura 8. Variables “bucket” origen

Para obtener la URL del archivo cargado en el “bucket” se vuelve a utilizar la información de la entrada del evento, luego se decodifican sus caracteres usando la función “`unquote_plus()`” que proviene de la biblioteca “`urllib.parse`”, y se

utiliza para facilitar el procesamiento. Además, para evitar problemas con las rutas de archivos, se eliminan los símbolos "/" como se muestra en la Figura 9.

```
for record in event['Records']:
  source_bucket = record['s3']['bucket']['name']
  destination_bucket = 'bucket-destino-erupcion-solar'
  key = unquote_plus(record['s3']['object']['key'])
  tmpkey = key.replace('/', '')
```

Figura 9. Variables con el nombre del archivo

Después se extrae el nombre del archivo subido al "bucket" y su extensión con la función "`splitext()`", para posteriormente poder construir los nombres de los archivos de salida (tanto la imagen como los metadatos en formato JSON), tal y como se ve en la Figura 10.

```
base_name, extension = os.path.splitext(tmpkey)
output_key = f"{base_name}_output{extension}"
json_key = f"metadata-{base_name}.json"
```

Figura 10. Archivos de salida

También es necesario establecer el nombre de la ruta temporal para guardar la imagen subida, y se utiliza la función "`uuid4()`" para generar un nombre de archivo único (ver Figura 11). Además, también se crean los nombres de las rutas en las que se subirán los archivos de salida ("`upload_image_path`" y "`upload_json_path`").

```
download_path = '/tmp/{}{}'.format(uuid.uuid4(), tmpkey)
upload_image_path = '/tmp/processed-{}'.format(tmpkey)
upload_json_path = '/tmp/metadata-{}.json'.format(tmpkey)
```

Figura 11. Rutas de archivos de salida

Una vez están listas todas las rutas temporales ya se puede proceder a descargar el archivo subido al "bucket" en las rutas establecidas. Para eso, primero se ha de inicializar un cliente para Amazon S3, lo cual se hace como en la Figura 12, utilizando la función "`client()`" de boto3, creando una variable global "`s3_client`". Tras esto se puede utilizar "`download_file()`" de la Figura 13, que recibe los parámetros "`source_bucket`" donde se especifica el nombre del "bucket" en S3, "`tmpkey`" que es el nombre del archivo subido al "bucket", y "`download_path`" que es la ruta local establecida en el paso anterior.

```
s3_client = boto3.client('s3')
```

Figura 12. Inicialización de cliente S3

```
s3_client.download_file(source_bucket, tmpkey, download_path)
```

Figura 13. Función download_file

Por último, antes de empezar el procesamiento de las imágenes, se abre el archivo descargado con "`with Image.open()`" para asegurar que se gestione correctamente la apertura y el cierre del mismo, además, una vez abierto, se guarda el archivo en la ruta temporal de salida con la función "`save()`" tal y como se ve en la Figura 14.

```
with Image.open(download_path) as img:
    img.save(upload_image_path)
```

Figura 14. Apertura segura de la imagen

4.4.1.3. Función circle_detection()

Para comenzar con la idea principal del algoritmo, se intenta detectar la longitud del radio del Sol en la imagen con la función "`circle_detection()`" que recibe la ruta de la imagen subida al "bucket" (ver Figura 15) y devuelve el valor deseado.

```
sun_circle_pixel_radius = circle_detection(download_path)
```

Figura 15. Función circle_detection

Dentro de la función se abre la imagen guardada en la ruta con la función "`imread()`" de OpenCV, después se convierte la imagen a escala de grises con "`cvtColor()`", que recibe por parámetro la imagen abierta anteriormente y el tipo de conversión de color que se realizará (en este caso "`cv2.COLOR_BGR2GRAY`" establece el cambio del formato de color estándar a un formato de color con únicamente tonos de gris). Por último, antes de proceder a la detección del radio del círculo solar, se aplica un desenfoque para reducir el ruido utilizando "`medianBlur()`" con 2 parámetros, el primero con la imagen convertida a escala de grises, y otro con un valor que a efectos prácticos representa la intensidad del desenfoque. Todo esto se puede observar en la Figura 16.

```python
def circle_detection(image_path):
    cv2_image = cv2.imread(image_path)
    cv2_grey_image = cv2.cvtColor(cv2_image, cv2.COLOR_BGR2GRAY)
    cv2_grey_image = cv2.medianBlur(cv2_grey_image, 5)
```

Figura 16. Transformaciones de la imagen

Tras haber hecho las transformaciones necesarias en la imagen, se aplica la función encargada de detectar círculos "`HoughCircles()`" para encontrar el disco solar. Se reciben 8 parámetros de entrada como se ve en la Figura 17, y sus valores se ajustan para detectar únicamente un gran círculo en toda la imagen, ya que solo interesa la detección del Sol. Los parámetros son los siguientes:

1. "`cv2_grey_image`": la imagen en la que se detectarán los círculos.
2. "`cv2.HOUGH_GRADIENT`": método de detección de círculos que utiliza el gradiente. Este gradiente es un valor que mide los cambios de intensidad de luz entre unos píxeles y otros para detectar los bordes.
3. "`dp`": un factor de escala que afecta a la precisión de la detección. El valor 1.2 se utiliza para mejorar el rendimiento en imágenes grandes.
4. "`minDist`": la distancia mínima entre los centros de los círculos detectados. Con valores suficientemente altos se evitan solapamientos.
5. "`param1`": un valor umbral para el detector de bordes de Canny (algoritmo utilizado para la detección de bordes en imágenes basado en el gradiente de las mismas), que cuanto mayor sea, menos bordes detectará, pero más intensos.
6. "`param2`": este segundo valor umbral tiene una finalidad similar al anterior.
7. "`minRadius`": el radio mínimo de los círculos que se deben detectar.
8. "`maxRadius`": el radio máximo de los círculos a detectar.

```python
circle_detected = cv2.HoughCircles(cv2_grey_image,
                                   cv2.HOUGH_GRADIENT,
                                   dp=1.2,
                                   minDist=1000,
                                   param1=70,
                                   param2=25,
                                   minRadius=150,
                                   maxRadius=250)
```

Figura 17. Función HoughCircles

En la variable "`circle_detected`" se guardan las coordenadas de los centros de los círculos detectados y la longitud del radio de estos. Por lo tanto a continuación se comprueba si se ha detectado algún círculo, se selecciona el conjunto de

círculos usando "`circle_detected[0, :]`" y se redondean los datos almacenados a números enteros con "`round().astype()`". Finalmente se devuelve el valor del radio corregido para no perder información en los próximos pasos. En caso de no detectar círculos se devuelve un valor por defecto que coincide con la longitud del radio del Sol de la mayoría de imágenes procesadas durante la validación del algoritmo. Cabe mencionar que no se utilizan las coordenadas del centro del círculo detectado debido a la falta de exactitud comprobada en la validación. Por lo tanto, en su lugar se utilizan las coordenadas del centro de la imagen, ya que el Sol siempre está centrado en las fotografías. Todos estos pasos se pueden ver tanto en la Figura 18 como en la Figura 19.

```
if circle_detected is not None:
    circle_detected = np.round(circle_detected[0, :]).astype("int")
    return circle_detected[0][2] - CIRCLE_PIXEL_RADIUS_CORRECTION
return DEFAULT_SUN_CIRCLE_PIXEL_RADIUS
```

Figura 18. Obtención del radio del sol

```
DEFAULT_SUN_CIRCLE_PIXEL_RADIUS = 202
CIRCLE_PIXEL_RADIUS_CORRECTION = 3
```

Figura 19. Constantes de radio por defecto y corrección de radio

El resultado de la detección de círculos para las imágenes que se utilizan en el proyecto, extraídas del SDO, sería el que se muestra en la Figura 20.

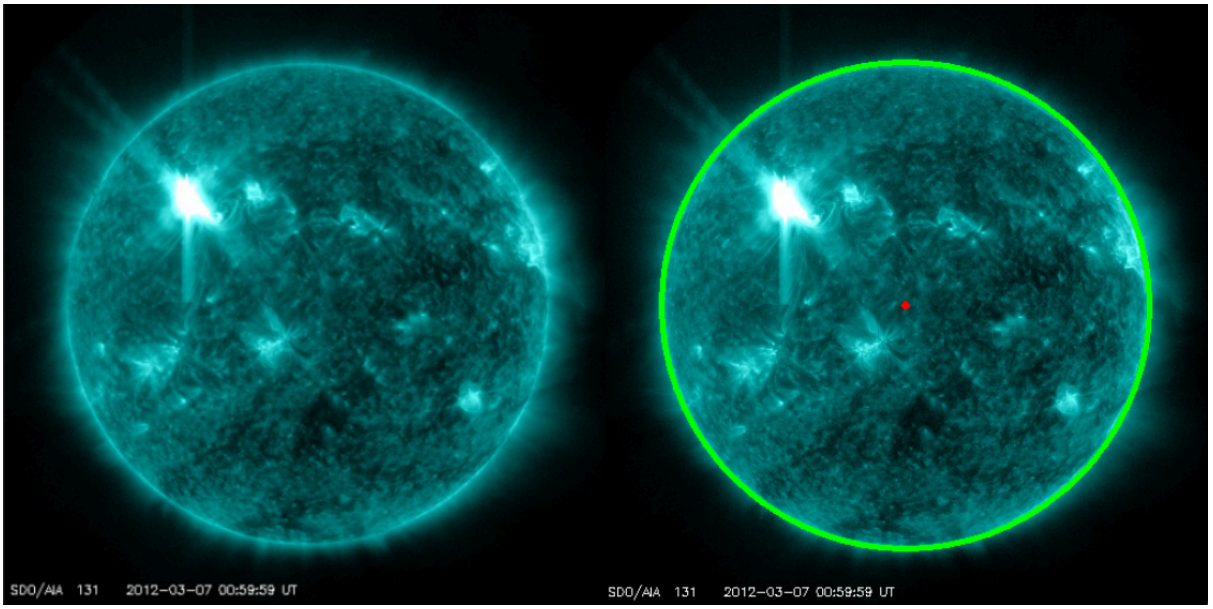

Figura 20. Detección de círculos ejemplificada

4.4.1.4. Función create_mask_image()

Una vez obtenido el radio del sol, se lleva a cabo el dibujado de la máscara con la función "`create_mask_image()`" que recibe el propio radio (ver Figura 21).

```
mask_image = create_mask_image(sun_circle_pixel_radius)
```

Figura 21. Función de creación de la máscara

Al inicio de la función se utiliza el método "`new()`" para crear una nueva imagen desde cero (ver Figura 22). Se reciben 3 parámetros de entrada, primero el modo de color de la imagen, en este caso "`RGBA`", que utiliza todos los colores y le añade la componente de transparencia. Después el tamaño de la máscara, que está definido en una constante (ver Figura 23) cuyo valor coincide con las dimensiones de las imágenes. Por último, se recibe el color inicial de todos los píxeles, que se establece como transparente.

```
def create_mask_image(sun_circle_pixel_radius):
    mask_image = Image.new("RGBA", MASK_SIZE, (0,0,0,0))
```

Figura 22. Creación de la nueva imagen para la máscara

```
MASK_SIZE = (512, 512)
```

Figura 23. Tamaño de la máscara

Justo después de crear esta máscara que por el momento es transparente, se crea un objeto para poder dibujar en ella mediante "`Draw()`" que simplemente recibe la propia máscara creada como parámetro tal y como se puede ver en la Figura 24.

```
mask_circle = ImageDraw.Draw(mask_image)
```

Figura 24. Función Draw

A continuación llega la hora de dibujar la máscara para cubrir las partes de la imagen que no aportan información relevante o incluso afectan negativamente a la detección (se explicará más en detalle tras repasar el código de esta sección). Para eso se utiliza "`rectangle()`" en caso de necesitar dibujar rectángulos, únicamente hacen falta las coordenadas y el color de relleno del dibujo. En la Figura 25 se observan múltiples ejemplos de esto.

```
mask_circle.rectangle([6, 490, 48, 504], fill="black")
mask_circle.rectangle([62, 494, 79, 501], fill="black")
mask_circle.rectangle([103, 494, 174, 501], fill="black")
mask_circle.rectangle([181, 494, 227, 501], fill="black")
mask_circle.rectangle([235, 494, 246, 501], fill="black")
```

Figura 25. Creación de rectángulos para la máscara

La última parte de la máscara es el círculo que tapa el sol, que se puede dibujar tras establecer las coordenadas necesarias y el color, negro y opaco en este caso. Para esto se llama a la función "`ellipse()`" como se ve en la Figura 26.

```
circle_center = (MASK_SIZE[0] // 2, MASK_SIZE[1] // 2)
mask_circle.ellipse((circle_center[0] - sun_circle_pixel_radius,
                     circle_center[1] - sun_circle_pixel_radius,
                     circle_center[0] + sun_circle_pixel_radius,
                     circle_center[1] + sun_circle_pixel_radius),
                     fill=(0,0,0,255))
```

Figura 26. Creación de la máscara circular

Se debe recalcar la importancia de dibujar la máscara para tapar los datos relacionados con el telescopio y la hora que se pueden ver en la parte inferior izquierda de las imágenes. En caso de no dibujarse puede afectar muy negativamente a la precisión del algoritmo de detección de erupciones más adelante. Es por esto que se crean tantos rectángulos diminutos, para ajustarse lo máximo posible al texto y no perder otra información relevante. En la Figura 27 se puede observar claramente el resultado de la imagen con esta máscara aplicada (se pintan de color rojo para facilitar su visibilidad en la memoria, pero en el proyecto real su color es negro). También es importante aclarar que de momento solo se tiene una imagen con fondo transparente y los dibujos mencionados anteriormente. Lo que se ve en la Figura 27 es el resultado que se obtendrá después de solapar la máscara generada con la imagen original, pero para ejemplificar el resultado de este paso se considera que es mejor ver la imagen de esta forma.

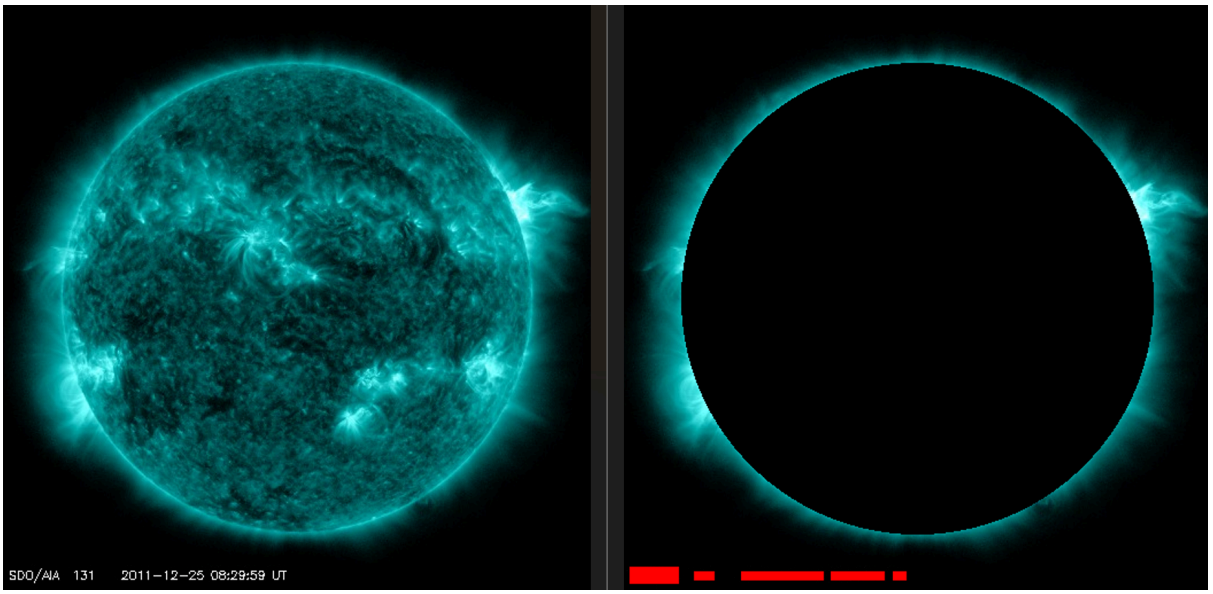

Figura 27. Máscara aplicada sobre la imagen

Para finalizar con la función "`create_mask_image`" se devuelve la imagen creada con la máscara, que se encuentra guardada en "`mask_image`" tal y como se ve en la Figura 28.

```
return mask_image
```

Figura 28. La imagen de la máscara es devuelta por la función

4.4.1.5. Función overlap_images()

Para proceder al solapamiento de la imagen, antes se abre la imagen subida al "bucket" utilizando "`with Image.open()`" para asegurarse de que se abra y se cierre correctamente. A continuación, como se puede ver en la Figura 29, se llama a la función "`overlap_images()`", cuyos parámetros de entrada son la máscara creada anteriormente y la imagen recién abierta.

```
with Image.open(download_path) as overlapped_image:
    overlap_images(mask_image, overlapped_image)
```

Figura 29. Función overlap_images

Dentro de la función, el primer paso que se debe realizar es convertir la imagen abierta a formato de color "`RGBA`" al igual que se hizo con la máscara en su creación, pero esta vez se usa el método "`convert()`". Justo después las

imágenes se encuentran listas para ser solapadas con la función "`paste()`" que recibe la primera imagen, la posición desde la que se solapará la segunda imagen (las coordenadas de la esquina superior izquierda), y la segunda imagen. Observar la Figura 30 para ver la implementación de la función "`overlap_images()`".

```
def overlap_images(mask_image, overlapped_image):
    overlapped_image.convert("RGBA")
    position = (0, 0)
    overlapped_image.paste(mask_image, position, mask_image)
```

Figura 30. Implementación de overlap_images

El estado de las imágenes en esta etapa del procesamiento es como el que se mostraba en la Figura 27 pero con los rectángulos de la máscara de color negro en vez de rojo.

4.4.1.6. Primer filtro

Una vez terminada la construcción de las imágenes se procederá a aplicar el primer filtro para detección de erupciones fuertes o que provocan un alto nivel de ruido visual en la imagen. Para esto se llama a la función "`first_filter_white_pixels_percentage()`", que tal y como se ve en la Figura 31, recibe como parámetro la imagen a procesar.

```
first_filter_percentage = first_filter_white_pixels_percentage(overlapped_image)
```

Figura 31. Llamada a la función del primer filtro

Este primer filtro utiliza el método "`convert()`" ya mencionado anteriormente, pero esta vez para transformar la imagen a binario con la configuración por defecto tal y como se muestra en la Figura 32. Esta transformación a binario se encarga de convertir todos los píxeles de la imagen a blanco o negro en función de lo claros u oscuros que sean. La configuración por defecto de la función no es muy restrictiva para considerar los píxeles blancos, por lo tanto es ideal en este primer filtro.

```
low_threshold_binary_image = overlapped_image.convert("1")
```

Figura 32. Conversión a binario

Los siguientes pasos tienen como finalidad averiguar el porcentaje de píxeles blancos en la imagen binaria. Para esto, primero se obtiene una lista con la información de los píxeles de la imagen con "`getdata()`". A continuación se

recorren todos los píxeles y se cuentan los de color blanco con el código de la Figura 33, y por último se obtiene el total de píxeles de la imagen con "`len()`". Una vez se ha obtenido esta información se devuelve el resultado de dividir el número de píxeles blancos entre el total de píxeles de la imagen y multiplicar esto por 100 como se ve en la Figura 34 para así obtener el porcentaje de píxeles blancos que se deseaba en un inicio.

```
first_filter_pixels = low_threshold_binary_image.getdata()
first_filter_white_pixels = sum(1 for pixel in first_filter_pixels if pixel == 255)
first_filter_total_pixels = len(first_filter_pixels)
```

Figura 33. Obtención de información de la imagen

```
return (first_filter_white_pixels/first_filter_total_pixels)*100
```

Figura 34. Porcentaje devuelto por la función

Tras devolver este valor, se almacena en "`first_filter_percentage`", y a continuación se comprueba si supera el umbral establecido para la detección de imágenes con ruido visual, que está guardado en la constante "`FIRST_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD`" de la Figura 35. Esta comprobación se lleva a cabo con un "`if()`" como el de la Figura 36.

```
FIRST_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD = 5.05
```

Figura 35. Constante de valor umbral del primer filtro

```
first_filter_percentage = first_filter_white_pixels_percentage(overlapped_image)
if (first_filter_percentage >= FIRST_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD):
```

Figura 36. Comprobación del primer filtro

En caso de que el porcentaje de píxeles blancos sea superior al valor umbral, significa que se ha detectado una imagen que contiene un gran número de píxeles blancos o con gran intensidad de ruido visual debido al deslumbramiento de una erupción solar. Por tanto, se guardará la imagen inicialmente subida al "bucket" de origen en el "bucket" destino, utilizando la función "`upload_file()`" que se ve en la Figura 37, el cual recibe, la ruta de archivo temporal, el "bucket" destino y el nombre que se dará al archivo. Todos estos datos recibidos son los que se crearon al inicio de la función "`lambda_handler()`".

```
s3_client.upload_file(upload_image_path, destination_bucket, output_key)
```

Figura 37. Función upload_file

Llegados a este punto, solo quedaría crear el archivo JSON con los metadatos de la imagen. Primero se deben preparar los datos que se escribirán en formato JSON. en este caso, como se puede ver en la Figura 38, se guarda en la variable "`detection`" el filtro en el que se ha detectado la erupción y en "`white_pixels_percentage`" el porcentaje de píxeles blancos detectados en este filtro. Es importante escribir el filtro en el que se detecta, ya que los porcentajes de píxeles blancos varían mucho entre el primer filtro y el segundo (que se explicará más adelante en la sección 4.4.1.8). Toda esta información se inserta en la variable "`metadata`", que es un diccionario que posteriormente será insertado en la ruta del archivo temporal creada anteriormente mediante "`with open()`", que abre el archivo en modo de escritura, y "`dump()`" que escribe en formato JSON la información dada en el archivo elegido, tal y como se muestra en la Figura 39. Finalmente se repite el uso de la función "`upload_file()`" para guardar el archivo creado en el "bucket", utilizando el nombre especificado al inicio en "`json_key`", y concluye la ejecución de la función Lambda.

```
detection = "filtro_1"
white_pixels_percentage = first_filter_percentage
metadata = {"detection": detection, "white_pixels_percentage": white_pixels_percentage}
```

Figura 38. Datos a guardar en JSON

```
with open(upload_json_path, 'w') as json_file:
    json.dump(metadata, json_file)
s3_client.upload_file(upload_json_path, destination_bucket, json_key)
```

Figura 39. Carga del archivo JSON en el "bucket" destino

4.4.1.7. Ejemplos de aplicación del primer filtro

Antes de pasar a explicar el resto del código, se ejemplificarán distintos casos relacionados con el primer filtro.

Primero, en la Figura 40 se puede ver el caso de una erupción solar que provoca mucho ruido en la imagen debido al deslumbramiento al momento de tomar la foto. Este sería un caso en el que el filtro detecta una erupción en la imagen debido a la cantidad de ruido generado.

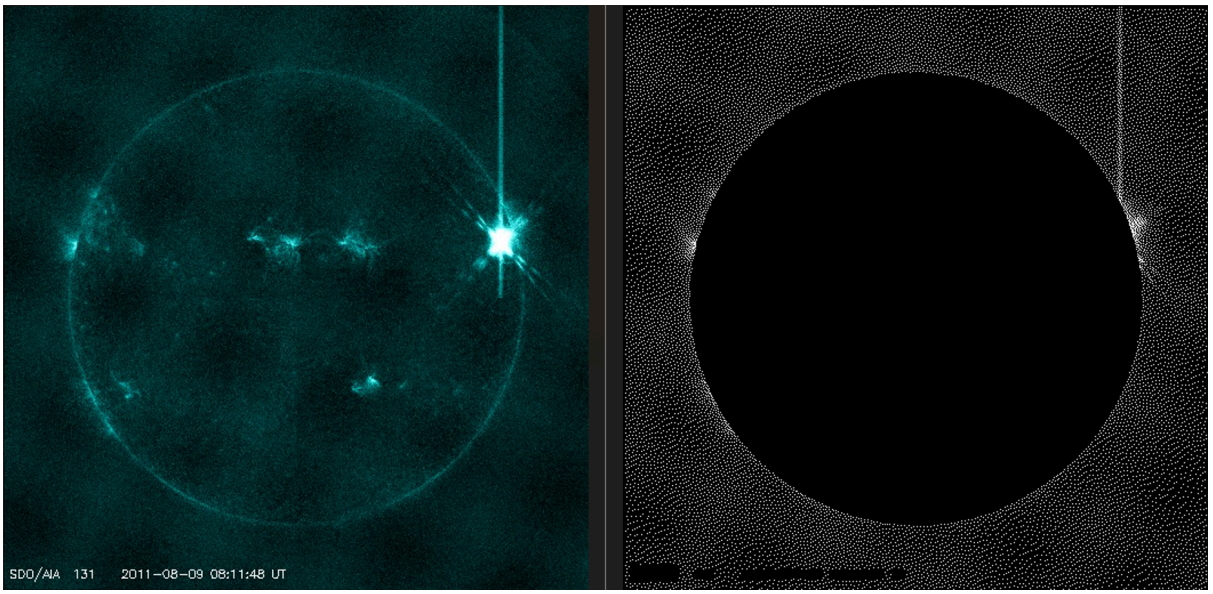

Figura 40. Ejemplo con alto índice de ruido en la imagen

En segundo lugar, la Figura 41 es un caso de una erupción solar que no provoca deslumbramiento, y por lo tanto no provoca ruido, pero al ser tan potente y generar tanta luz, es detectada en el primer filtro.

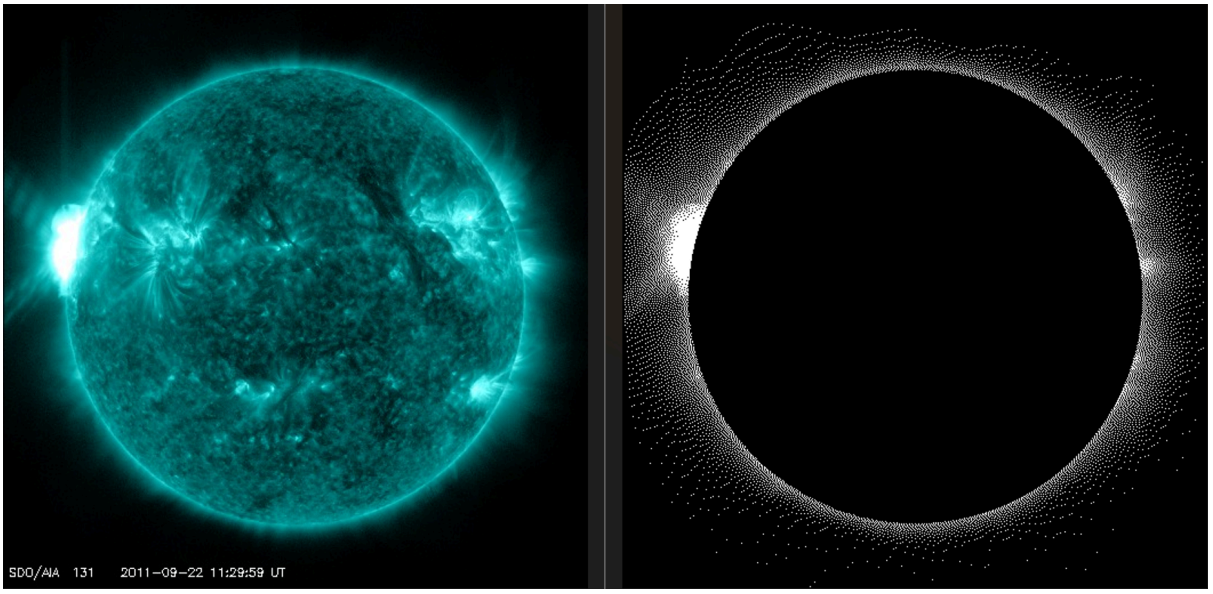

Figura 41. Ejemplo erupción solar sin ruido de imagen

El último ejemplo de la Figura 42 es el caso de una erupción interior que provoca deslumbramiento y genera algo de ruido. El problema de esta erupción es que no genera suficiente ruido visual como para ser considerada una erupción, y si el valor umbral fuese más bajo y esta imagen pasase el filtro, en consecuencia se encontrarían muchos falsos positivos. Por lo tanto, la detección de estas erupciones

que se dan en el interior del disco solar es un aspecto que se podría mejorar en el trabajo de cara al futuro.

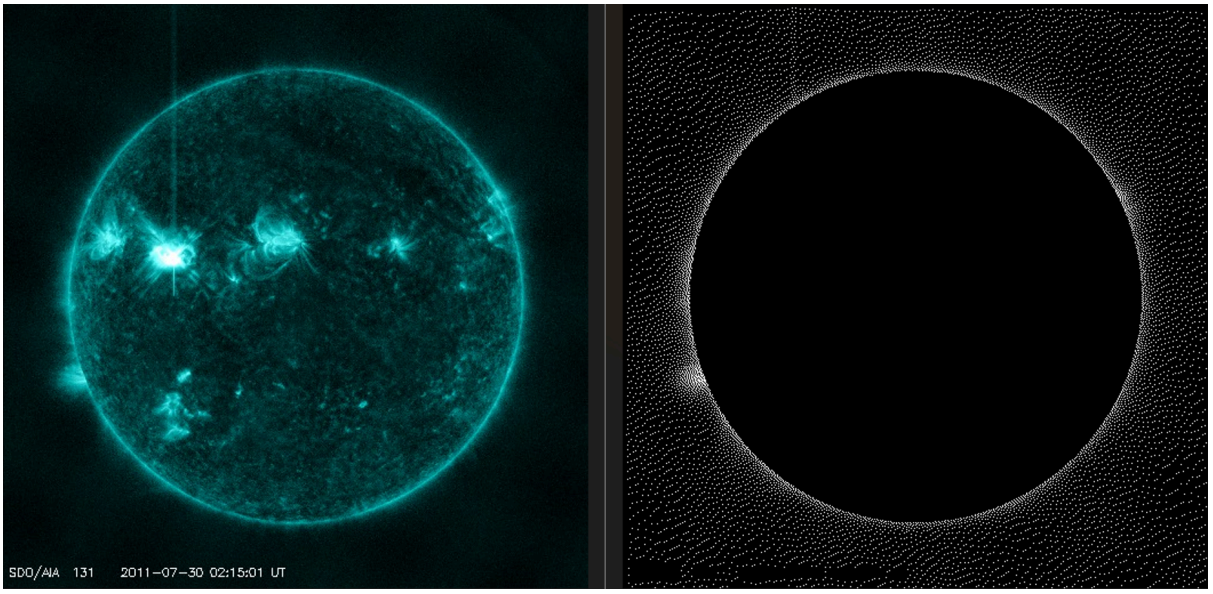

Figura 42. Ejemplo con bajo índice de ruido en la imagen

4.4.1.8. Segundo filtro

Es el momento de analizar el camino que sigue el código en caso de no haber detectado un porcentaje de píxeles blancos satisfactorio en el primer filtro. Para esto se entra en la condición "`else`" del condicional de la Figura 36, que empieza con la llamada a la función "`second_filter_white_pixels_percentage`" de la Figura 43, que aplica el segundo filtro y recibe el mismo parámetro que el primero.

```
else:
    second_filter_percentage = second_filter_white_pixels_percentage(overlapped_image)
```

Figura 43. Llamada a la función del segundo filtro

Este segundo filtro es algo más extenso que el anterior, ya que se necesita filtrar los píxeles blancos de una forma más restrictiva, es decir, que para considerar un píxel blanco en vez de negro, este debe tener una gran intensidad de luz. Para conseguir crear un filtro más restrictivo es necesario añadir algunos pasos extra con respecto al primer filtro. En concreto hay que evaluar los píxeles uno a uno, por lo que en vez de convertir la imagen a binario directamente, se convertirá a escala de grises de forma que se pueda analizar la intensidad de blanco de cada pixel en el rango de 0 (negro) a 255 (blanco) que se utiliza por defecto.

A continuación se explicará el código utilizado para conseguir realizar estos pasos. Primero se transforma la imagen recibida por parámetro a escala de grises con "`convert()`", y después se utiliza "`point()`" para aplicar una transformación a cada pixel de la imagen, pasándole una función anónima "`lambda`" con el método que se seguirá. Para cada pixel, se comprobará si su intensidad de blanco es superior a un valor umbral establecido en la constante "`PIXEL_LIGHT_INTENSITY_THRESHOLD`". En caso de superarlo, se establece el valor del píxel a 255, y en caso contrario se pone a 0. Estos pasos se llevan a cabo en el código de la Figura 44.

```
def second_filter_white_pixels_percentage(overlapped_image):
    grey_image = overlapped_image.convert("L")
    high_threshold_binary_image = grey_image.point(
        lambda p: 255 if p > PIXEL_LIGHT_INTENSITY_THRESHOLD else 0, "1")
```

Figura 44. Transformación a binario de la imagen

El siguiente paso es el mismo que se realiza en el primer filtro: se suma el total de píxeles blancos en la imagen y el total de píxeles de cualquier color. Una vez se obtienen estos datos, se devuelve el porcentaje de píxeles blancos de este filtro como se puede ver en la Figura 45.

```
second_filter_white_pixels = sum(1 for pixel in second_filter_pixels if pixel == 255)
second_filter_total_pixels = len(second_filter_pixels)
return (second_filter_white_pixels/second_filter_total_pixels)*100
```

Figura 45. Cálculo del porcentaje de píxeles blancos

De vuelta en la función "`lambda_handler`", se comprueba si el valor devuelto por el segundo filtro es superior al valor umbral establecido en la variable "`SECOND_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD`" en la Figura 46.

Por un lado, en caso de superarse el valor umbral significa que se ha detectado una erupción, por lo que se repiten los mismos pasos que en el primer filtro, la subida de la imagen original al "bucket" destino, la generación de los metadatos y su subida en formato JSON en el "bucket" destino. La única diferencia es la adición de un dato nuevo "`intensity`", cuya función es dar una predicción del tipo de erupción que se ha dado según el porcentaje de píxeles blancos detectados. Para poder diferenciar las erupciones fuertes, medias y débiles se utilizan las constantes "`SECOND_FILTER_STRONG_SOLAR_FLARE_THRESHOLD`" (para distinguir las erupciones solares fuertes de las de intensidad media) y "`SECOND_FILTER_MEDIUM_SOLAR_FLARE_THRESHOLD`" (para distinguir las

erupciones medias de las débiles) como se puede ver en la Figura 47. El código asociado a todo lo mencionado es el de la Figura 48.

Por otro lado, si el porcentaje de píxeles blancos detectados en la imagen es inferior al valor umbral del filtro, la ejecución de la función Lambda termina sin generar archivos de salida.

```
SECOND_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD = 0.00939
```

Figura 46. Constante con el valor umbral del segundo filtro

```
SECOND_FILTER_MEDIUM_SOLAR_FLARE_THRESHOLD = 0.04
SECOND_FILTER_STRONG_SOLAR_FLARE_THRESHOLD = 0.06
```

Figura 47. Constantes con valores umbral de clasificación

```
if (second_filter_percentage >= SECOND_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD):
    s3_client.upload_file(upload_image_path, destination_bucket, output_key)
    detection = "filtro_2"
    white_pixels_percentage = second_filter_percentage
    intensity = "weak"
    if(white_pixels_percentage > SECOND_FILTER_STRONG_SOLAR_FLARE_THRESHOLD):
        intensity = "strong"
    elif (white_pixels_percentage >= SECOND_FILTER_MEDIUM_SOLAR_FLARE_THRESHOLD):
        intensity = "medium"
    metadata = {"detection": detection, "white_pixels_percentage": white_pixels_percentage, "intensity": intensity}
    with open(upload_json_path, 'w') as json_file:
        json.dump(metadata, json_file)
    s3_client.upload_file(upload_json_path, destination_bucket, json_key)
```

Figura 48. Preparación de archivos de salida del segundo filtro

4.4.1.9. Ejemplos de aplicación del segundo filtro

De forma similar al primer filtro, se analizarán algunos ejemplos aplicados en el segundo filtro para terminar de entender su funcionamiento.

En el primer ejemplo, en la Figura 49, se puede observar una erupción solar de intensidad media, tras su transformación a binario de la forma ya explicada anteriormente, solo se ve una pequeña mancha blanca en la imagen, pero es de un tamaño lo bastante grande como para ser considerada una erupción solar.

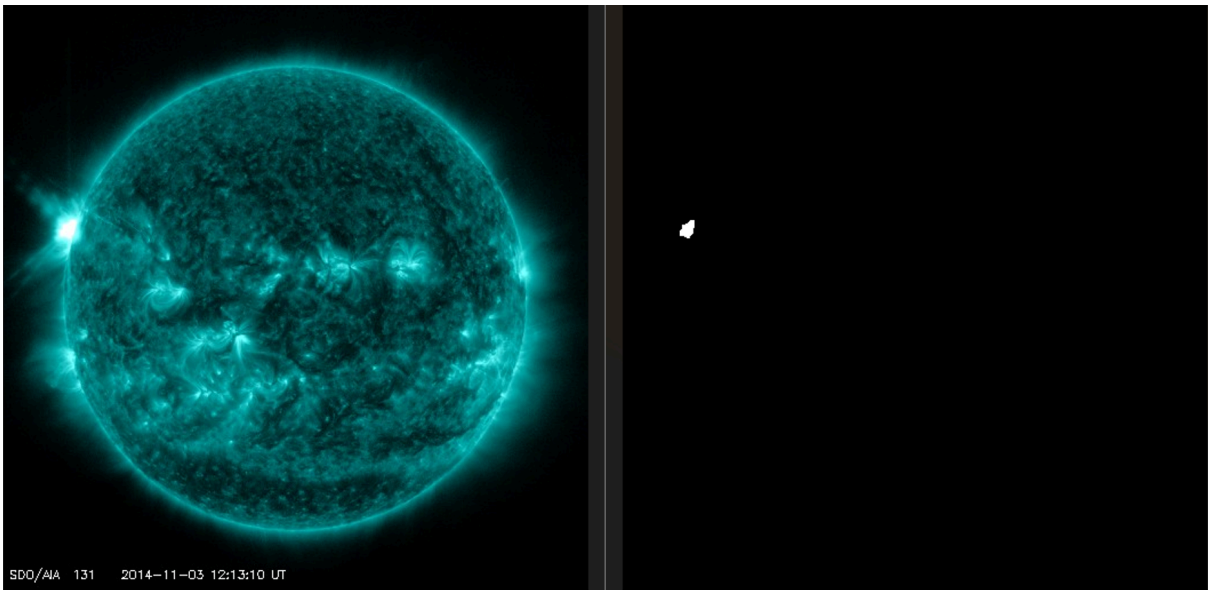

Figura 49. Ejemplo de erupción de intensidad media con segundo filtro aplicado

En este otro caso de la Figura 50, la erupción solar es bastante débil (de hecho es considerada erupción solar por el filtro con un margen extremadamente bajo). Por eso mismo en la imagen binaria la mancha que representa la luz de la erupción apenas se puede apreciar.

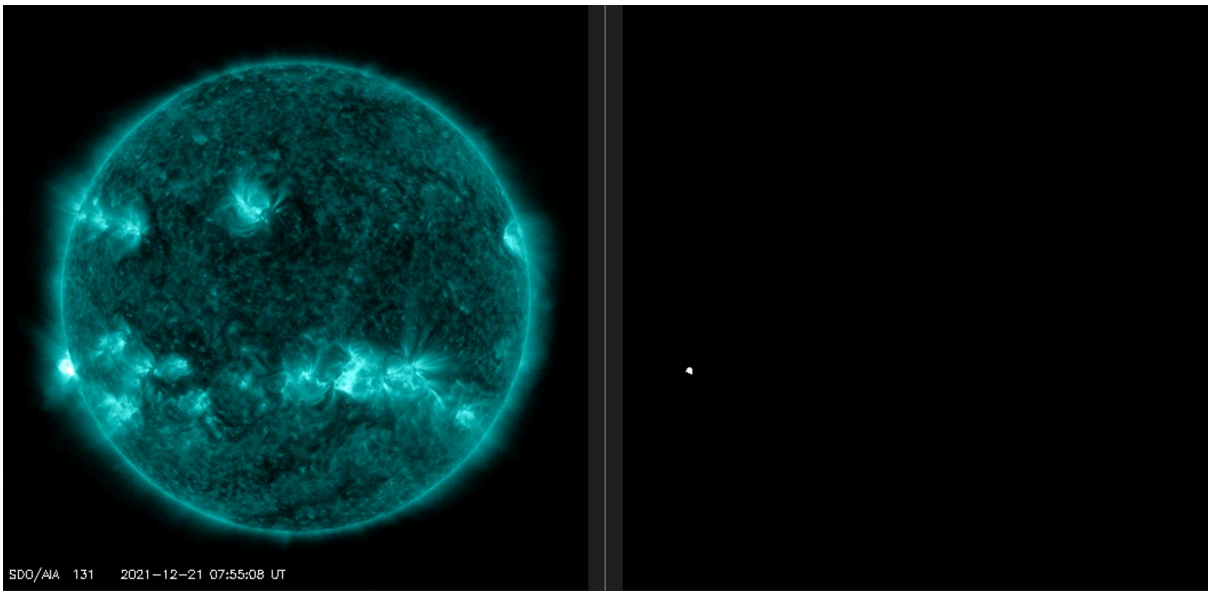

Figura 50. Ejemplo de erupción de intensidad baja con segundo filtro aplicado

Para terminar con los ejemplos, la Figura 51 es el caso de un intento de falso positivo, ya que aunque no hay una erupción solar, el Sol emite más luz de lo habitual en un punto, por lo que podría engañar al filtro y ser procesada generando un caso de falso positivo. Sin embargo, el valor umbral del porcentaje de píxeles

blancos es ligeramente superior al de esta imagen, precisamente para evitar falsos positivos con imágenes similares a esta.

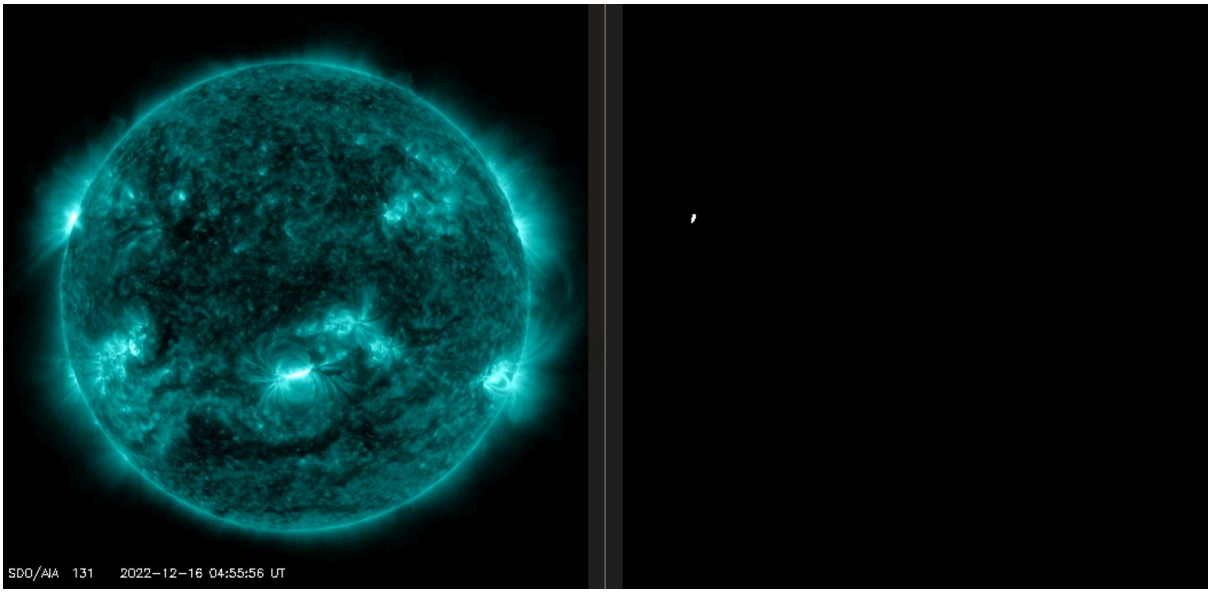

Figura 51. Ejemplo de posible falso positivo con segundo filtro aplicado

4.4.2 Obtención de valores umbral

Este apartado se utilizará para explicar de dónde provienen los valores asignados a las constantes que guardan los umbrales tanto de los filtros, como de clasificación o cualquier otro.

Para hallar el valor umbral de las constantes utilizadas en los filtros, que sirven para saber si un porcentaje de píxeles es suficiente como para considerar que existe una erupción en esa imagen, se realizó un estudio empírico basado en el análisis del conjunto de imágenes disponibles. Se necesitaron 2 tipos de imágenes: por un lado, imágenes con una alta intensidad de luz emitida pero sin erupciones presentes (a las que se podría llamar "falsos positivos"). Y por otro lado, imágenes con erupciones solares de baja o muy baja intensidad.

Con estas imágenes ya se pudo establecer una cota superior para el porcentaje de píxeles blancos en imágenes sin erupciones que pudieran constituir “falsos positivos”, y a la vez, una cota inferior de las imágenes con erupciones de baja intensidad. Una vez encontradas estas cotas, se pudo establecer un valor umbral que incluyese a las erupciones de baja intensidad pero no a los casos de “falsos positivos”. Utilizando este proceso se guardaron los valores óptimos en las constantes “`FIRST_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD`” (para el

primer filtro) y “SECOND_FILTER_WHITE_PIXELS_PERCENTAGE_THRESHOLD” (para el segundo filtro) en el código.

En el caso de los valores umbral utilizados para clasificar las erupciones según su intensidad (débil, intermedia y fuerte), se recolectaron todas las imágenes con erupciones solares que se daban en el exterior del disco solar, se filtraron en el algoritmo para guardar todas las que no fuesen detectadas en el primer filtro y se ordenaron por la intensidad real de las erupciones a las que estaban asociadas (clasificadas con el formato de letra y número, X13, X2, M6, etc…). Una vez realizados estos pasos, se tenía una lista con la intensidad de las erupciones y el porcentaje de píxeles blancos hallados en la imagen con el segundo filtro.

El principal problema de la relación entre ambos valores era la falta de imágenes, ya que aunque se disponía de imágenes asociadas a las distintas erupciones, estas fotos muchas veces no habían sido tomadas en sus momentos más intensos. Esto se debía a que el telescopio encargado de tomar las imágenes tiene una frecuencia de una foto cada 15 minutos aproximadamente, pero en otros casos no tomaba fotos en varias horas, por lo que las imágenes disponibles para algunas erupciones no resultaban muy representativas con respecto a lo que habían sido.

Esto provocaba que algunas erupciones de intensidad M5 cuya foto había sido tomada en el momento exacto en el que tuvo lugar, tuviesen un porcentaje de píxeles blancos superior a erupciones X2 debido a que su foto había sido tomada transcurridos 20 minutos del origen de la erupción.

Volviendo con la lista obtenida en el paso anterior, primero se clasificaron las distintas intensidades de las erupciones obtenidas en base al estándar del NOAA. A continuación, se estableció el rango de porcentajes de píxeles blancos entre los que se encontraban las imágenes de las erupciones para cada intensidad (M1, M2, X1, etc…). Finalmente se halló una media aproximada con todos los casos de las distintas intensidades. En la Figura 52 se pueden ver los 3 campos mencionados representados en las distintas columnas. Además, debajo se pueden apreciar los valores umbral obtenidos en base a los datos.

	(a)	(b)
M1	0,009-0,016	~0,01
M2	0,013-0,085	~0,03
M3	0,012-0,074	~0,03
M4	0,011-0,093	~0,04
M5	0,012-0,078	~0,04
M6	0,010-0,090	~0,02
M7	0,012-0,083	~0,05
M8	0,011-0,054	~0,02
M9	0,019-0,054	~0,04
X1	0,012-0,231	~0,06
X2	0,027-0,132	~0,08
X3	0,039-0,084	~0,07
X4	0,065-0,065	~0,065
X5	0,084	~0,084
X7	0,070	~0,07

débil: % < 0,04
media: 0,04 <= % <= 0,06
fuerte: 0,06 < %

Figura 52. Categorización de los diferentes tipos de erupciones solares estudiadas (según el estándar de clasificación del NOAA) y valores típicos de: a) rangos de porcentajes de píxeles blancos en las imágenes de cada conjunto y b) valores medios

Los valores umbral obtenidos con el método explicado se guardaron en las constantes "`SECOND_FILTER_MEDIUM_SOLAR_FLARE_THRESHOLD`" y "`SECOND_FILTER_STRONG_SOLAR_FLARE_THRESHOLD`".

CAPÍTULO 5. MANUAL DE USUARIO Y PREPARACIÓN DEL ENTORNO

En esta sección se explicará cómo configurar el entorno para que cualquier usuario pueda reproducir lo expuesto en la memoria del proyecto.

El primer paso será la creación de una cuenta de AWS, que es un paso sencillo pero cada usuario deberá crearla con sus propios datos.

5.1. “Buckets” S3

Una vez se disponga de una cuenta de AWS y se haya iniciado sesión, será necesario crear los “buckets” con los que se trabajará, tanto el de origen, como el de destino. Para esto simplemente se debe utilizar el buscador en la propia página de AWS y escribir S3 como en la Figura 53.

Figura 53. Búsqueda de S3

Una vez seleccionada la opción de S3, se redirigirá a la página de Amazon S3. El primer paso será cambiar en la parte superior derecha de la pantalla la región, y poner la localización más cercana siguiendo la idea de la Figura 54.

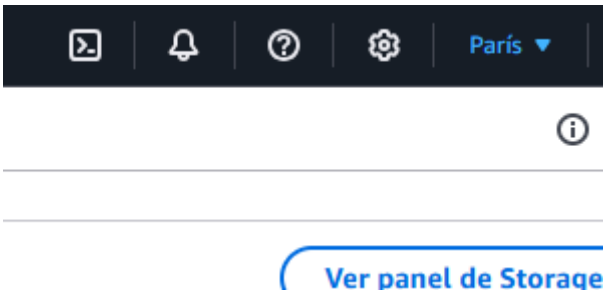

Figura 54. Región AWS

Una vez elegida la ubicación correctamente, se procederá a crear los “buckets”. Para ello se debe seleccionar la opción “Crear bucket” que aparece resaltado en naranja en la Figura 55.

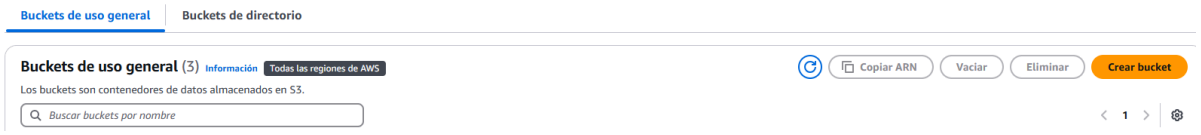

Figura 55. Opción de creación de “bucket”

Dentro de la ventana de creación, se debe elegir un nombre único para el “bucket”, en este proyecto se eligió “bucket-origen-erupcion-solar”. Además se debe asegurar que en el apartado “Región de AWS” esté la ubicación geográfica más cercana como en la Figura 56.

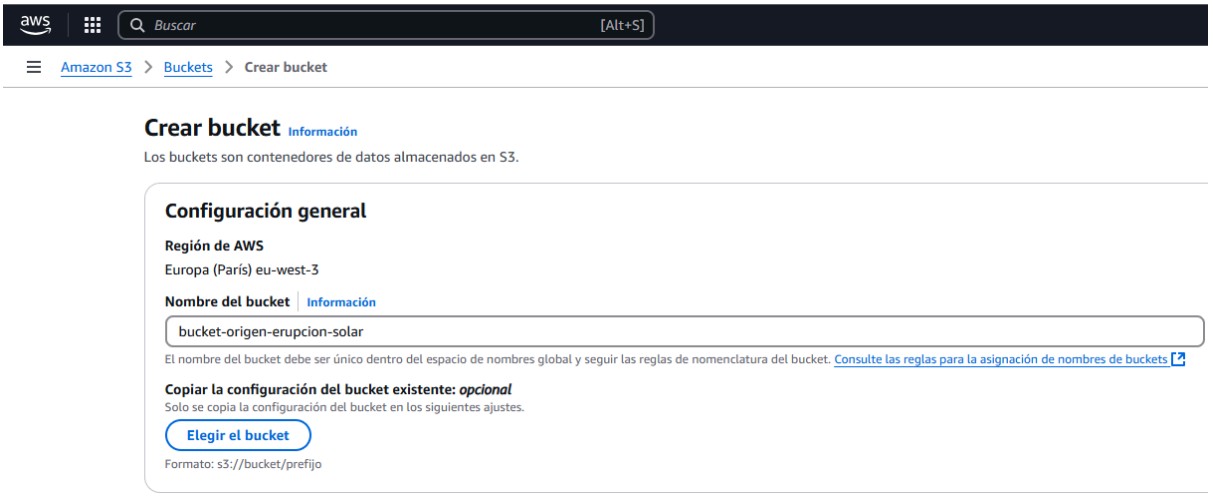

Figura 56. Ventana de creación de “buckets”

El resto de opciones se dejarán con sus valores por defecto y luego se pulsará la opción de “Crear bucket” en naranja de abajo a la derecha. Se debe repetir el mismo proceso para el “bucket” destino, solo que cambiándole el nombre al que se desee.

También será necesario crear un tercer “bucket” para guardar el paquete de despliegue que se generará más adelante, este se creará de la misma forma que se han creado los anteriores.

Para subir un archivo al “bucket” creado, se debe hacer click en el “bucket”, dentro de la ventana de “buckets”, que se verá como la Figura 57 y luego seleccionar “Cargar” en la nueva ventana de la Figura 58. Por último se debe apretar el botón “Agregar archivos” y seleccionar el archivo con la imagen de la erupción solar que se quiera tratar, para después pulsar “Cargar” en naranja, abajo a la derecha (ver Figura 59).

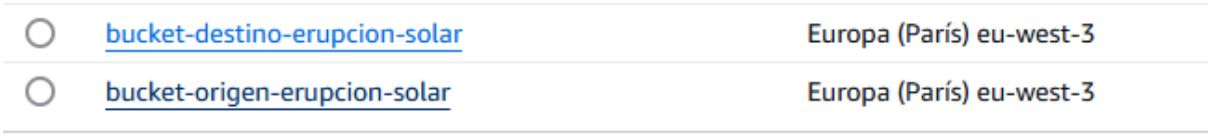

Figura 57. “Buckets” creados

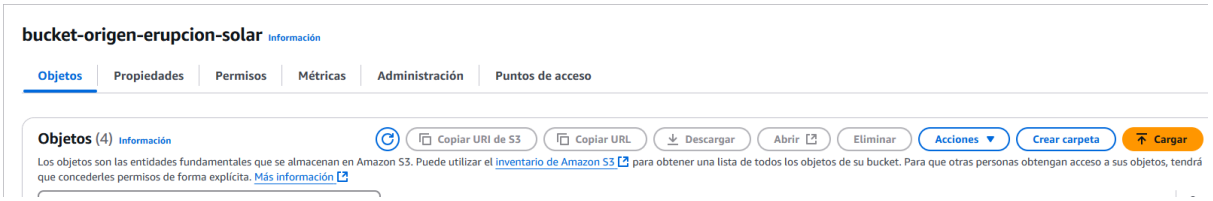

Figura 58. Cargado de imágenes

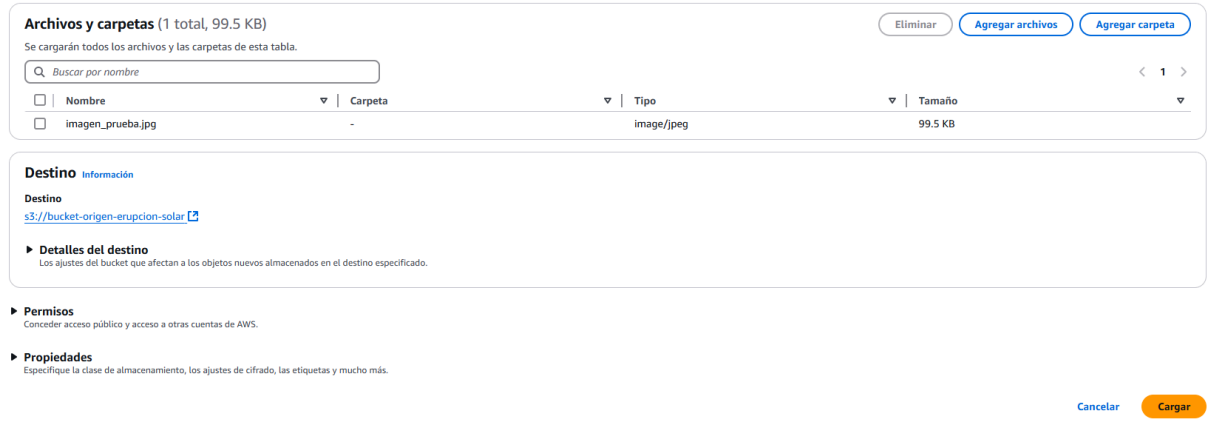

Figura 59. Selección de archivos

5.2. Política de permisos

El siguiente paso será la creación de la política de permisos para poder crear la función Lambda más tarde. Para esto se deben seguir los pasos de la Figura 60,

escribiendo IAM en el buscador y seleccionando la opción “Políticas” del desplegado.

Figura 60. Búsqueda de Políticas

Una vez se accede a la página, se debe seleccionar la opción “Crear política” de la esquina superior derecha. Dentro de la nueva pestaña, a la altura de “Editor de políticas” estará seleccionada la opción “Visual”, pero se debe elegir “JSON” y pegar el siguiente código (extraído de un ejemplo con funcionalidad similar de la página oficial de AWS)[30]:

```
{
    "Version": "2012-10-17",
    "Statement": [
        {
            "Effect": "Allow",
            "Action": [
                "logs:PutLogEvents",
                "logs:CreateLogGroup",
                "logs:CreateLogStream"
            ],
            "Resource": "arn:aws:logs:*:*:*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:GetObject"
            ],
```

30 https://docs.aws.amazon.com/es_es/lambda/latest/dg/with-s3-tutorial.html#with-s3-tutorial-create-policy

```
            "Resource": "arn:aws:s3:::*/*"
        },
        {
            "Effect": "Allow",
            "Action": [
                "s3:PutObject"
            ],
            "Resource": "arn:aws:s3:::*/*"
        }
    ]
}
```

Una vez pegado el código proveniente de la página oficial de AWS, debería verse como en la Figura 61.

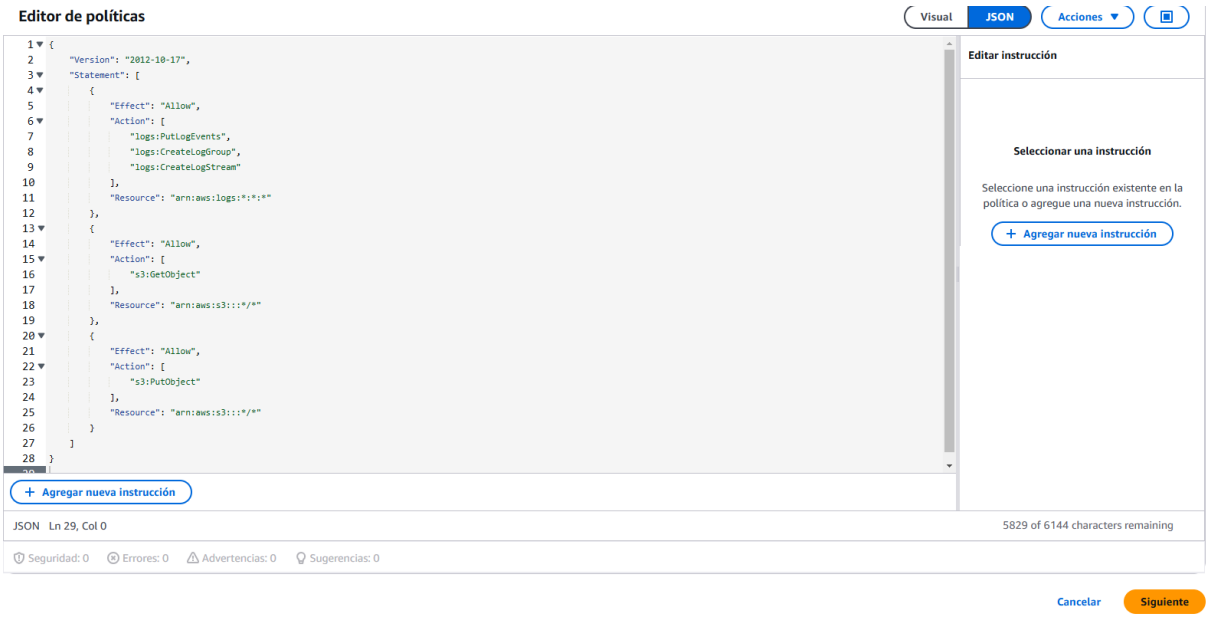

Figura 61. Código de políticas

Por lo que se seleccionará “Siguiente”, abajo a la derecha. A continuación solo habrá que elegir el nombre de la política, “LambdaS3Policy” por ejemplo, y “Crear política”, abajo a la derecha de nuevo.

5.3. Rol de ejecución

Después de la política de permisos toca crear un rol de ejecución, que utilizará la función Lambda para acceder a los servicios y recursos de AWS. De nuevo se

escribirá “IAM” en el buscador, pero esta vez se elegirá la opción “Roles” como en la Figura 62.

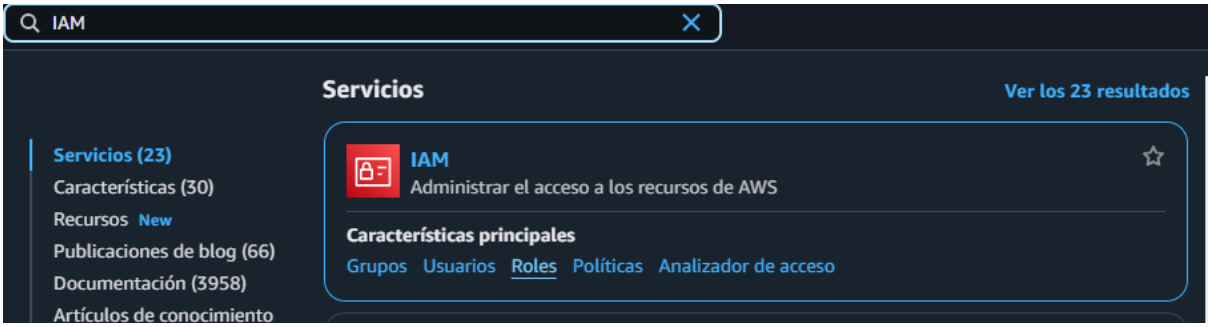

Figura 62. Búsqueda de Roles

En la nueva ventana se utilizará la opción en naranja de “Crear rol” que se puede ver en la Figura 63.

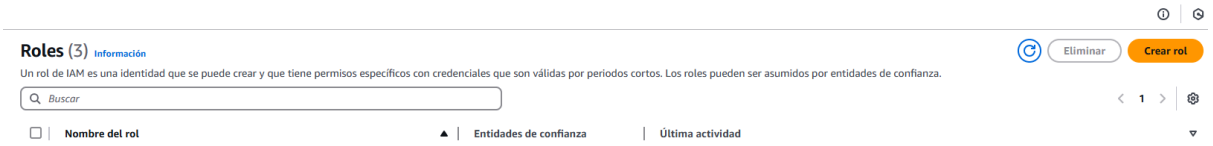

Figura 63. Creación de rol

A continuación se elegirá la opción “Servicio de AWS” si es que no viene ya por defecto, y se buscará la opción “Lambda” en el apartado “Caso de Uso”, el resultado debe verse como la Figura 64.

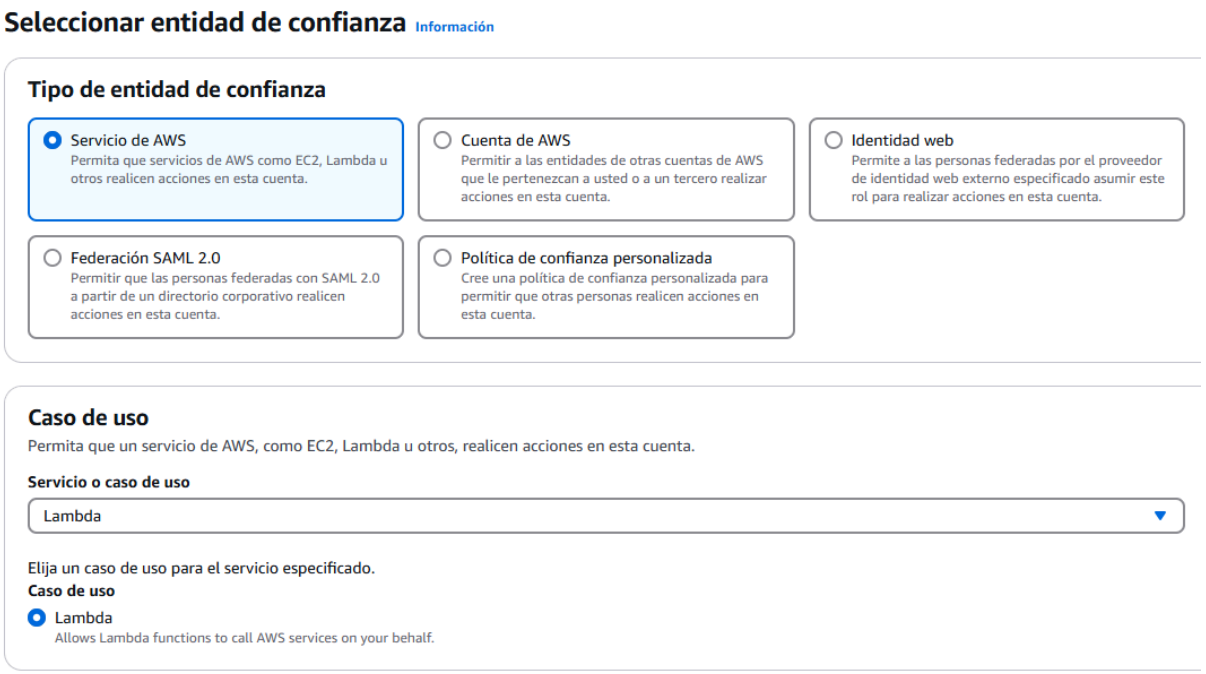

Figura 64. Opciones de creación de rol de ejecución

El siguiente paso será buscar el nombre de la política de permisos creada anteriormente y seleccionar la casilla izquierda como muestra la Figura 65, y después pulsar “Siguiente”.

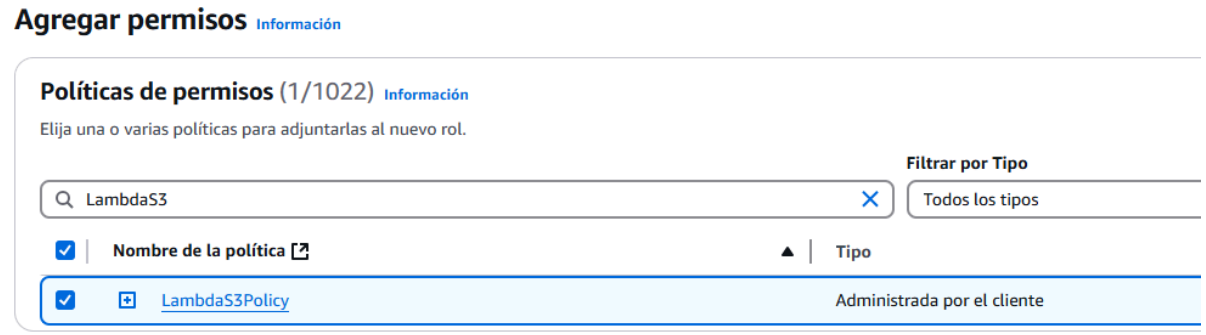

Figura 65. Selección de políticas de permisos

El último paso solo requerirá escribir el nombre del rol, por ejemplo, “LambdaS3Policy” y de nuevo seleccionar el botón naranja de creación.

5.4. Paquete de despliegue

Antes de crear la función es necesario crear el paquete de despliegue, que es un archivo que contiene todo el código fuente y las dependencias necesarias para poder ejecutar la función.

Lo primero es guardar el código del algoritmo en Python con el nombre que se desee. En el mismo directorio en el que se guarde el archivo, se debe crear un nuevo directorio llamado “package” por ejemplo. Utilizando por ejemplo la terminal de Visual Studio Code se debe acceder al directorio “package” y utilizar el comando:

```
pip install --platform manylinux2014_x86_64 --target package
--implementation cp --python-version 3.12 --only-binary=:all:
--upgrade numpy pillow opencv-python-headless
```

Este comando permitirá instalar las dependencias de las librerías utilizadas en el algoritmo (se recomienda utilizar la misma versión de Python, 3.12, para evitar posibles errores).

Una vez se han instalado las dependencias, se debe volver al directorio en el que están guardados el directorio “package” y el archivo con el código del algoritmo, para ejecutar el comando:

```
Compress-Archive -Path .\package\*, .\lambda_function.py
-DestinationPath .\lambda_function.zip
```

El archivo “lambda_function.py” es el nombre del archivo que contiene el algoritmo en Python, y “lambda_function.zip” el nombre del archivo comprimido que se generará. Tras esto se debería tener una carpeta con la estructura que se ve en la Figura 66.

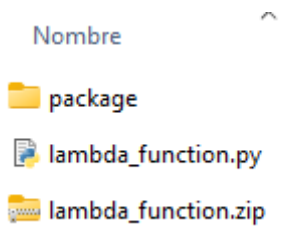

Figura 66. Estructura de carpetas

El archivo comprimido se subirá al tercer “bucket” que se creó al inicio cuyo fin era contener el paquete de despliegue.

5.5. Función Lambda

Ya se puede empezar la creación de la función en AWS Lambda, para lo que será necesario volver a la página principal de AWS y buscar Lambda, seleccionar la opción y rellenar los campos de la misma forma que en la Figura 67. En el nombre de la función elegir el que se desee, en el apartado “Tiempo de ejecución” seleccionar el lenguaje y versión utilizado, en este caso será “Python 3.12”, en “Arquitectura” la que viene por defecto, y en “Rol de ejecución” se debe elegir “Uso de un rol existente” y seleccionar el creado anteriormente “LambdaS3Role”. Una vez completado esto se puede crear la función.

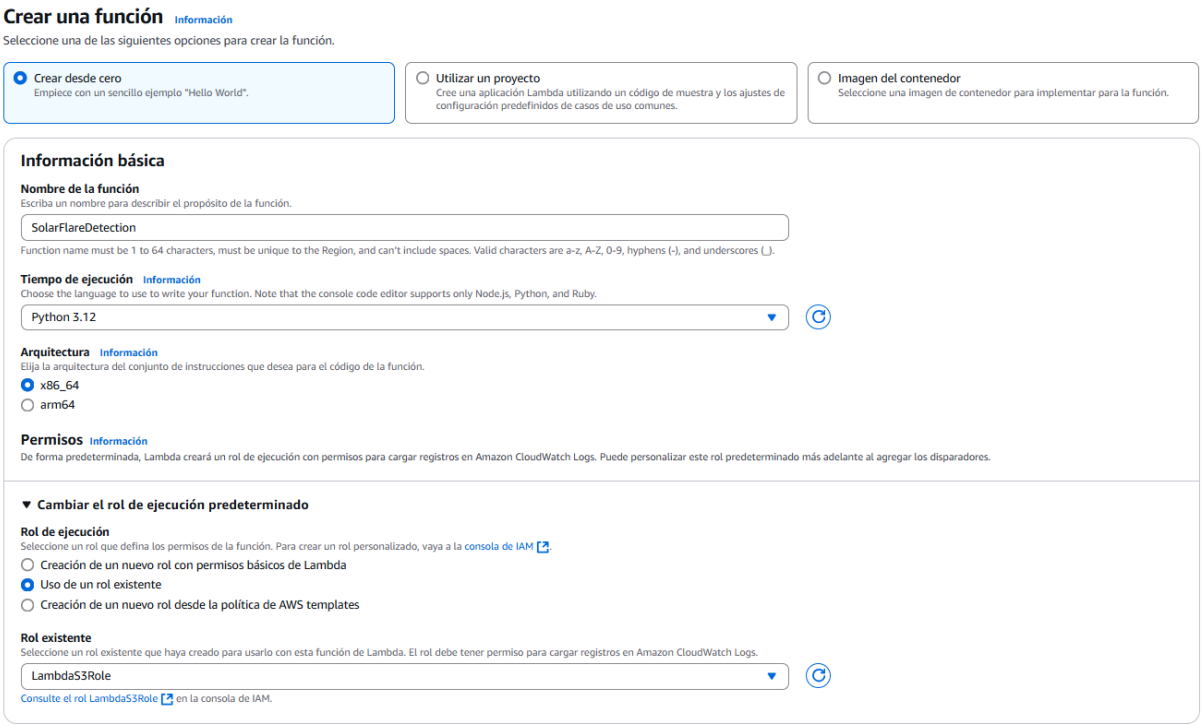

Figura 67. Opciones de creación de función lambda

Dentro de la función recién creada es necesario cargar el código del paquete de despliegue. Para esto se entrará en la función y se seleccionará el desplegable “Cargar desde”, y luego “Ubicación de Amazon S3” como en la Figura 68, lo que mostrará la ventana emergente de la Figura 69.

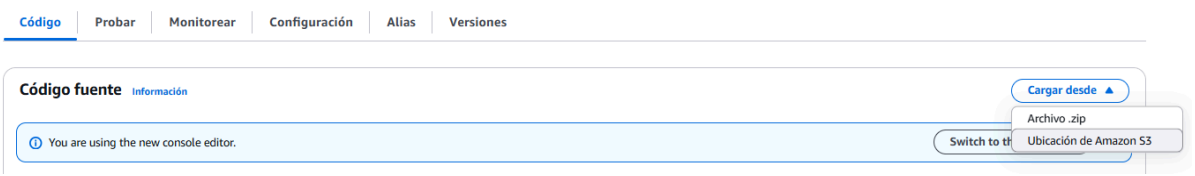

Figura 68. Carga del paquete de despliegue

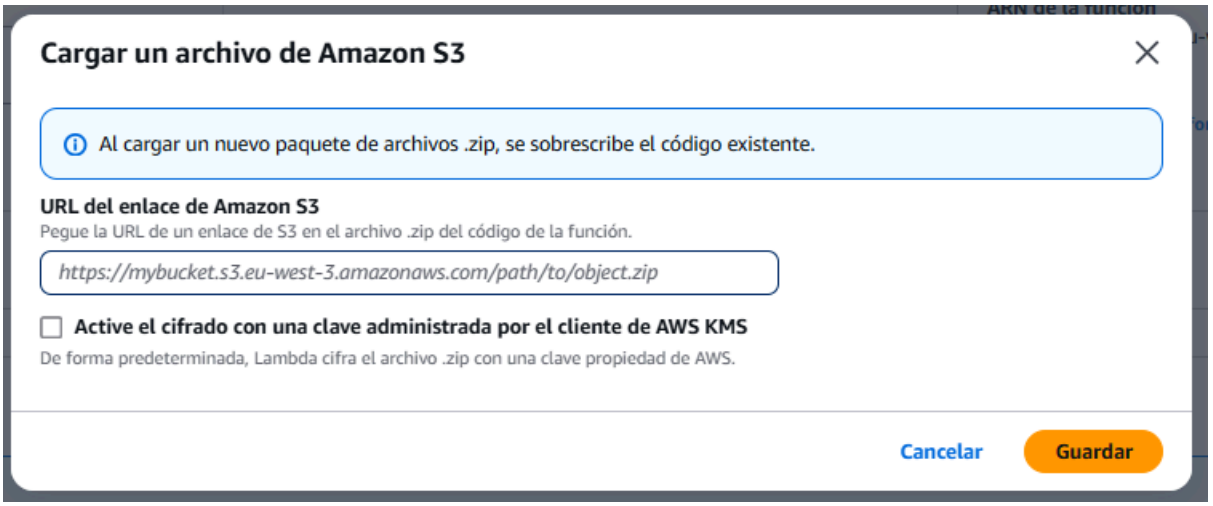

Figura 69. Inserción de URL del enlace de S3

La URL del enlace de Amazon S3 que se necesita se puede obtener yendo al "bucket" que contiene el paquete de despliegue, seleccionando la pestaña del comprimido y haciendo click en "Copiar URL" como se ve en la Figura 70.

Figura 70. Obtención de URL

Una vez seleccionada la opción "Guardar" de la Figura 69 ya se habría completado la creación de la función. La creación del "bucket" para contener el paquete de despliegue solo es necesaria debido al tamaño del archivo comprimido: si fuese más ligero podría cargarse directamente el comprimido en la función sin necesidad de crear un tercer "bucket".

5.6. Desencadenador

Tras la creación de la función el último paso de la configuración es el desencadenador, para que la función se active siempre que éste se lance. En este caso se configurará de forma que la función se ejecute cuando se suba un archivo al "bucket" de origen. Para esto se debe abrir la función creada y elegir "Agregar desencadenador" como en la Figura 71.

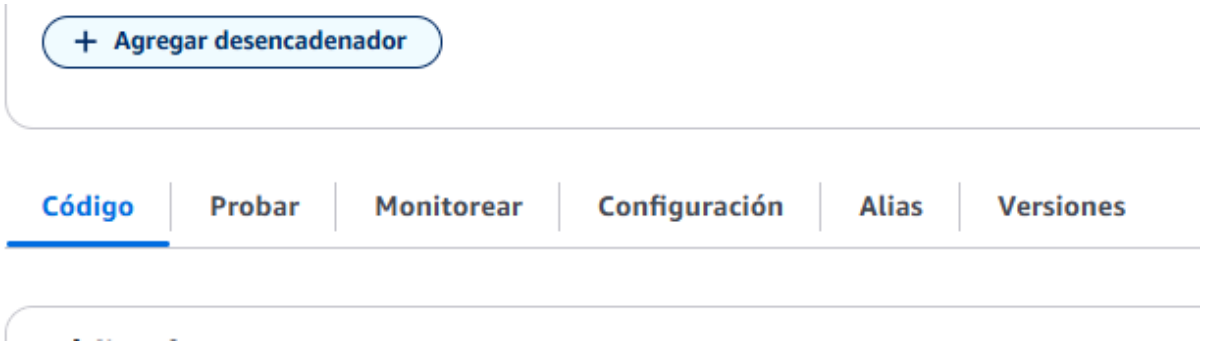

Figura 71. Agregación de desencadenador a la función

En "Configuración del desencadenador" escribir S3, luego elegir el "bucket" de origen, en la sección de "Tipos de eventos" elegir la que aparece por defecto de "Todos los eventos de creación de objetos", marcar la casilla de "Invocación recurrente", y pulsar el botón naranja de "Agregar" de la Figura 72.

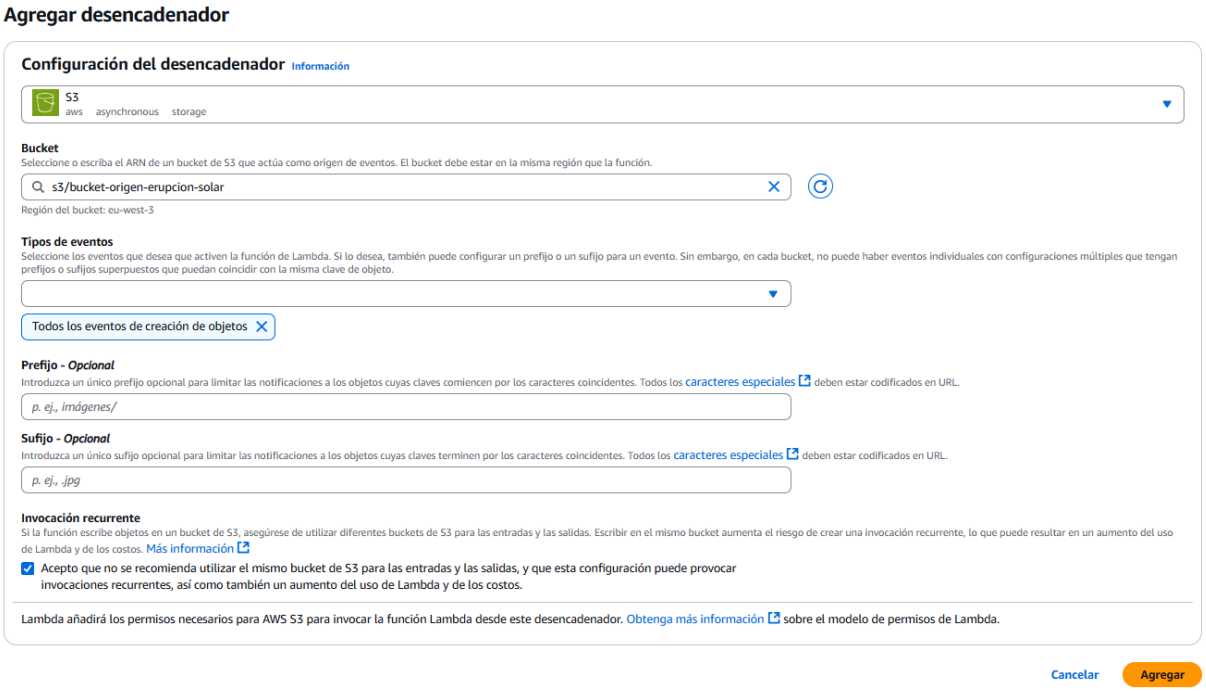

Figura 72. Opciones de configuración del desencadenador

En este punto la dinámica de subida de imágenes y detección de erupciones solares mediante la función Lambda ya debería funcionar. Sin embargo, se recomienda hacer un último paso para gestionar de forma más eficiente los recursos utilizados. Para esto será necesario entrar a la función creada e ir al apartado de "Configuración", a continuación se pulsa el botón "Editar" como en la Figura 73.

Figura 73. Edición de la función

En esta ventana de configuración se escribirá lo mismo que en la Figura 74, en "Memoria" 128 MB, "Almacenamiento efímero" 512 MB y "Tiempo de espera" de 15 segundos. Con estos parámetros se reducirán los recursos consumidos al mínimo para que funcione.

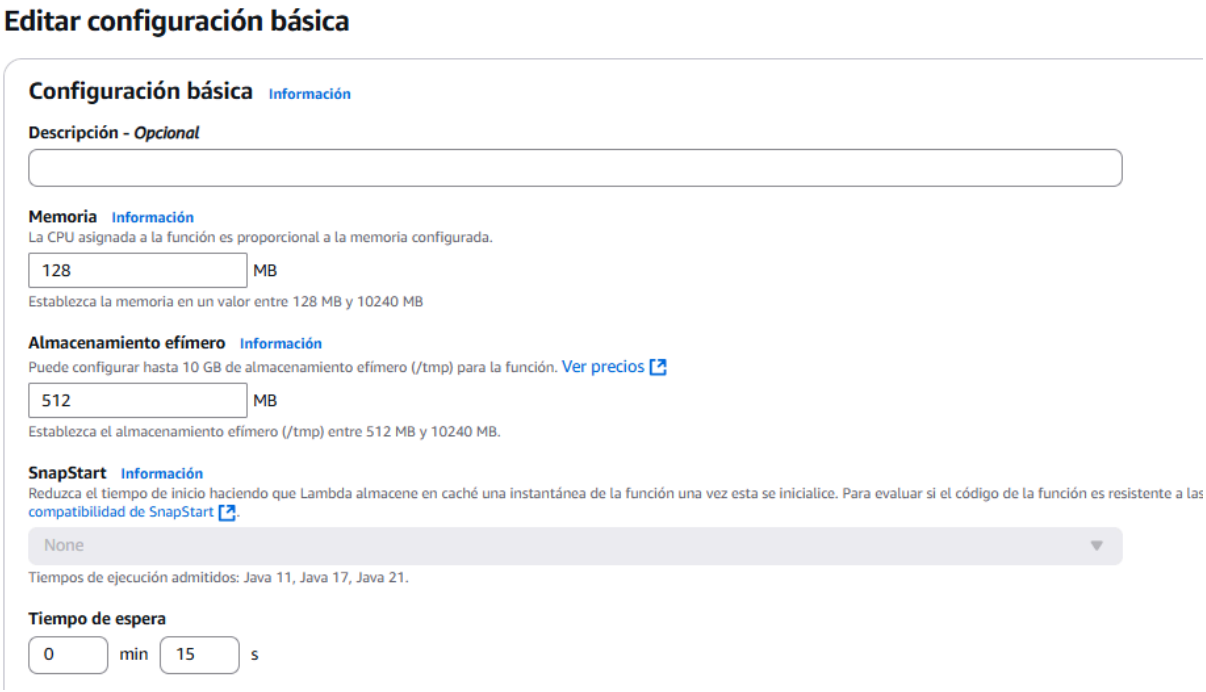

Figura 74. Edición de configuración de la función

CAPÍTULO 6. CONCLUSIONES Y TRABAJO FUTURO

Tras meses de trabajo se consiguió alcanzar la meta propuesta, crear un sistema de detección de erupciones solares utilizando una arquitectura basada en computación sin servidor.

Se encontró una fuente fiable de imágenes del Sol (el SDO) en las que se podían apreciar las erupciones solares. Después se creó una función Lambda con un algoritmo que actuaba en respuesta a la subida de imágenes en un “bucket” de S3.

La creación de este algoritmo permite la detección de erupciones solares a través de una serie de pasos en las imágenes subidas. Primero, se realiza una detección de círculos en la imagen para obtener el radio del disco solar. Después, se genera una imagen con siluetas negras dibujadas que se utiliza como máscara para tapar partes de las imágenes que no son útiles. A continuación, se superpone la máscara generada sobre la imagen original, y la fusión de estas da como resultado una sola imagen. Tras esto, la imagen ya está preparada para pasar los filtros de detección de erupciones solares. El primer filtro transforma la imagen a binario e intenta detectar erupciones solares en la imagen buscando el ruido visual producido por el deslumbramiento de estas. En caso de no detectar una erupción con el primer filtro se procede a utilizar el segundo. Este otro filtro también transforma la imagen a binario pero con un criterio mucho más estricto, por lo que un pequeño conjunto de píxeles blancos en la imagen transformada se interpreta como la aparición de una erupción solar.

Finalmente las imágenes con erupciones detectadas se guardan en un “bucket” de salida acompañadas de archivos de metadatos en formato JSON.

La principal conclusión a la que se ha podido llegar tras la elaboración del proyecto es la utilidad del procesamiento de imágenes para resolver problemas específicos. Al fin y al cabo, la parte más importante del algoritmo utilizado se basa únicamente en aplicar máscaras y filtros a la imagen para poder tratarla y así obtener datos relevantes.

Por otro lado, también se ha aprendido la importancia de utilizar una fuente de datos consistente en la que los datos sigan un mismo formato, y en concreto en el caso de las imágenes tengan el mismo tamaño, resolución y perspectiva. El motivo es que los más mínimos cambios pueden poner en peligro la efectividad del algoritmo. Un mínimo cambio en la fuente de la que se obtienen las imágenes, por ejemplo, una reducción de la intensidad de la luz de las mismas, una variación en el enfoque o en

la marca de agua con la información de la imagen, podrían dar lugar a falsos negativos o positivos.

Por último, cabe mencionar lo mucho que ha facilitado todo el trabajo el uso de una arquitectura basada en computación sin servidor, en concreto la parte enfocada a la preparación del entorno, ya que no es necesario configurar ni mantener servidores y se han podido centrar los esfuerzos en la realización del código y la extracción de imágenes útiles.

La eficacia del algoritmo diseñado es elevada para erupciones que se dan en el extremo del disco solar, debido a que su eyección de masa coronal se observa fácilmente. Sin embargo, cuando estas erupciones se dan en el interior del disco solar no hay forma de distinguirlas correctamente, ya que las imágenes están representadas en 2 dimensiones y no es posible observar si se está expulsando materia. Por lo tanto, una nueva forma de abordar el proyecto y extenderlo en un futuro podría ser la exploración de nuevos métodos para distinguir estas erupciones internas.

Por ejemplo, podría utilizarse algún modelo de machine learning específico para la detección de objetos en imágenes y entrenarse desde cero con los ejemplos de erupciones solares que se pueden extraer de la página del SDO utilizada en este proyecto. En caso de necesitar más imágenes de prueba o validación, se podría incluso aplicar alguna técnica de incremento de imágenes que transforman sutilmente las ya existentes aplicando distintos filtros.

Para este modelo de “machine learning” se podrían explorar tecnologías en la nube como la ofrecida por AWS, Amazon SageMaker, que permite integrar modelos de aprendizaje automático y puede resultar muy práctica dado el uso de AWS en el proyecto.

CHAPTER 6. CONCLUSIONS AND FUTURE WORK

After months of work, the proposed goal of creating a solar flare detection system using a serverless computing architecture was achieved.

It was possible to find a reliable source of images of the sun (the SDO) where solar flares could be observed. A Lambda function was then created with an algorithm that worked in response to the uploaded images in an S3 bucket.

The creation of this algorithm allows solar flares detection in the uploaded images in a series of steps. First, a circle detection is performed on the image to get the solar disc radius. Then an image with black shapes drawn on it is generated and used as a mask to hide some parts of the images that are not useful. The generated mask is then overlaid on the top of the original image, and the merge of both images end up as a single image. After this, the image is ready to pass the solar flare detection filters. The first filter transforms the image to binary and tries to detect solar flares in the image by looking for visual noise produced by solar flare glare. If the first filter does not detect a flare, the second filter is used. This other filter transforms the image to binary too but with a much stricter standard, so that a small set of white pixels in the transformed image is interpreted as a solar flare.

Finally, the images with detected flares are saved in an output bucket along with metadata files in JSON format.

The main conclusion drawn from the project is the usefulness of image processing for solving specific problems. After all, the most important part of the algorithm used is based just on applying masks and filters to the image in order to process it and obtain relevant data.

On the other hand, it has also been learned the importance of using a consistent data source where the data keeps the same format, and in particular in the case of images they have the same size, resolution and perspective. The reason is that the slightest changes can endanger the effectiveness of the algorithm. A slight change in the source from which the images are obtained, for example, a reduction of the image light intensity, a variation in the focus or in the watermark with the image information, could lead to false negatives or false positives.

Finally, it is worth mentioning how much easier the use of a serverless computing architecture has made all the work, in particular the part focused on the preparation of the environment, since it is not necessary to configure or maintain servers, so it

has been possible to focus the efforts on building the code and the extraction of useful images.

The efficiency of the designed algorithm is high for eruptions occurring at the edge of the solar disc, because their coronal mass ejection is easily observed. However, when these eruptions take place inside the solar disc, there is no way to distinguish them correctly, as the images are represented in two dimensions and it is not possible to observe if the matter is being ejected. Therefore, a new way to approach the project and extend it in the future could be to explore new methods to distinguish these inner eruptions.

For example, a specific machine learning model for detecting objects in images could be used and trained from the start with solar flares examples that can be extracted from the NASA page used in this project. In case more test or validation images are needed, it is possible to apply some image augmentation techniques that subtly transform the existing ones by applying different filters.

For this machine learning model, one could explore cloud technologies such as Amazon SageMaker, offered by AWS, which allows the integration of machine learning models and can be very useful counting with the use of AWS in the project.

BIBLIOGRAFÍA

[1] Estefania Blanch LLosa y David Altadill Felip, Meteorología Espacial: Los Efectos de las Tormentas Solares a la Tierra, 2018. http://cienciaprop.fundaciocaixavinaros.com/wp-content/uploads/2018/08/paper13_Blanch_Altadill.pdf

[2] Baris Dincer, Solar Flares Prediction / Unsupervised Learning, 2021. https://www.kaggle.com/code/brsdincer/solar-flares-prediction-unsupervised-learning/notebook

[3] Krishnaprasad Chirakkil, Robert J. Lillis, Justin Deighan, Michael S. Chaffin, Sonal K. Jain, David A. Brain, Matthew O. Fillingim, Raghuram Susarla, Greg Holsclaw, Xiaohua Fang, Nick M. Schneider, Hoor AlMazmi, Hessa AlMatroushi, Marko Gacesa, Nayla El-Kork, Ed Thiemann, Jasper S. Halekas. EMM EMUS Observations of FUV Aurora on Mars: Dependence on Magnetic Topology, Local Time, and Season. Journal of Geophysical Research: Planets, Volume 129, Issue 6, article id. e2024JE008336 (2024). https://doi.org/10.1029/2024JE008336

[4] Ezquerro Moya, Pablo. Constructo: sistema serverless para la generación de topologías mediante Machine Learning. Trabajo de Fin de Grado de la Universidad Complutense de Madrid, Facultad de Informática (2024). https://hdl.handle.net/20.500.14352/107920

[5] Pacios, D., Vázquez-Poletti, J.L., Dhuri, D.B. et al. A serverless computing architecture for Martian aurora detection with the Emirates Mars Mission. Sci Rep 14, 3029 (2024). https://doi.org/10.1038/s41598-024-53492-4

[6] Edmond Erik Larsen, Machine Learning model survey with the dataset for solar flare prediction, 2021. https://commons.erau.edu/cgi/viewcontent.cgi?article=1218&context=db-srs