PROYECTO FIN DE GRADO

TÍTULO: Adaptación, despliegue y análisis de rendimiento del simulador SISIFO sobre arquitecturas serverless o de microservicios

AUTOR/A: Diego Muñoz Beltrán

TITULACIÓN: Grado en Ingeniería Telemática

TUTOR/A: Francisco Javier Ramírez Ledesma

DEPARTAMENTO: Dpto. de Ingeniería Telemática y Electrónica (DTE)

VºBº TUTOR/A

Miembros del Tribunal Calificador:

PRESIDENTE/A: Francisco José Arqués Orobón

TUTOR/A: Francisco Javier Ramírez Ledesma

SECRETARIO/A: Javier Martín Rueda

Fecha de lectura:

Calificación:

El Secretario/La Secretaria,

Agradecimientos

A Alfonso, Juan y Miki, porque han estado tanto tiempo pendientes de este trabajo como yo mismo. Gracias por ser mis amigos más cercanos tantos años después.

A Rafa, María y Fran, por enseñarme a confiar en mí mismo, a recuperar la ilusión académica y a entender cuál es el esfuerzo requerido para llegar hasta aquí.

A Guille, Jorge, Julio y Vittorio, por ser los compañeros de carrera y amigos que pensé que nunca iba a tener. Gracias por hacer el camino más fácil.

A Luis Narvarte y a mi tutor, Francisco Javier Ramírez. Al primero, por inspirarme durante todo un curso y permitirme reconciliarme y hacer las paces con esta universidad. Al segundo, por su buen hacer y su tiempo dedicado a lo largo de este proyecto, a veces en horas intempestivas. Gracias especialmente a los dos por mostrarme el amor por el mundo académico, que no siempre es fácil de ver en un aula, y la pasión por la investigación. Gracias por permitirme participar en un proyecto acorde con mis ideas, que ha puesto un bonito broche final a esta etapa.

A mis tíos Marisa y Javier, por ser como unos padres, los mejores que cualquier persona pueda tener, y por inculcarme los valores y las ganas de hacer un mundo más amable y justo. A mi abuela María Luisa, por su cariño y su solidaridad al acogerme en el peor momento. A los tres, infinitas gracias por vuestra bondad, paciencia y confianza. Gracias por permitirme desarrollarme como persona.

A Cristina Bushell, la persona que mejor me conoce y que me ha enseñado a conocerme y perdonarme. No estaría aquí sin ti.

A mi padre, Francisco Alfonso, la persona que lleva inspirándome toda la vida y a quien le debo todo, no solo la vida. Allá donde estés, gracias por esas largas e interminables horas viéndote delante de una pantalla, tu pasión es mi pasión.

En especial, al Diego de septiembre de 2007, el que pensó que no valía, que no era suficiente y que nunca iba a lograr llegar hasta este día. Te lo dedico, para que la próxima vez que leas esto sepas que se puede conseguir todo con constancia y voluntad.

Resumen

El proyecto **“Adaptación, despliegue y análisis de rendimiento del simulador SISIFO sobre arquitecturas serverless o de microservicios”** sigue la estela de una tendencia mundial: el uso de servicios distribuidos y arquitecturas basadas en *serverless*.

Serverless, como indica su nombre, es una modelo de arquitectura de infraestructura en la nube que permite a los desarrolladores diseñar y ejecutar aplicaciones sin necesidad de configurar o administrar servidores y recursos de infraestructura.

Este proyecto actúa como un caso de estudio en el que se adapta parte de una aplicación existente consistente en funciones con estructura monolítica a un modelo distribuido haciendo uso de una plataforma FaaS (Function as a Service).

Tras el despliegue de las funciones sobre la plataforma, se realizan diferentes pruebas por parte del autor para analizar el rendimiento y escalabilidad de la solución, y se comparan los resultados con los obtenidos con el modelo de implementación previa.

En las conclusiones se incluye una investigación sobre cómo aprovechar las ventajas de esta arquitectura para mejorar la eficiencia y reducir costos en futuros proyectos.

En última instancia, el objetivo del proyecto es contribuir al conocimiento y entendimiento de esta tecnología en evolución, así como proponer una posible vía de mejora y continuación del proyecto.

Abstract

The project **"Adaptation, deployment, and performance analysis of the SISIFO simulator on serverless or microservices architectures"** follows a global trend: the use of distributed services and serverless-based architectures.

Serverless, as its name suggests, is an architecture model for cloud infrastructure that allows developers to design and run applications without the need to configure or manage servers or infrastructure resources.

This project acts as a case study in which part of an existing application consisting of monolithic structured functions is adapted to a distributed model using a FaaS (Function as a Service) platform.

After deploying the functions on the platform, the author runs different tests to analyze the performance and scalability of the solution, and the results are compared with those obtained from the previous implementation model.

The conclusions include an investigation into how to leverage the advantages of this architecture to improve efficiency and reduce costs in future projects.

Finally, the goal of this project is to contribute to the knowledge and understanding of this evolving technology, as well as to propose a way for improving and developing the existing project.

Índice de contenidos

Índice de figuras y capturas de pantalla

Índice de tablas

Lista de acrónimos

Acrónimo	Descripción
AGPLv3	Licencia Pública General Affero versión 3 (Affero General Public License version 3)
AMPQ	Protocolo Avanzado de Cola de Mensajes (Advanced Message Queuing Protocol)
API	Interfaz de Programación de Aplicaciones (Application Programming Interface)
AWS	Amazon Web Services
AZ	Zona de Disponibilidad (Availability Zone)
CPD	Centro de Procesamiento de Datos
CPU	Unidad Central de Procesamiento (Central Processing Unit)
CRD	Definición de recurso personalizado (Custom Resource Definition)
CSS	Hojas de estilo en cascada (Cascading Style Sheets)
DNS	Sistema de Nombres de Dominio (Domain Name System)
EBS	Servicio de Bloqueo Elástico (Elastic Block Store)
EC2	Nube de Computación Elástica (Elastic Compute Cloud)
EKS	Servicio de Kubernetes de Amazon (Elastic Kubernetes Service)
ELB	Balanceador de Carga Elástico (Elastic Load Balancing)
FV	Panel fotovoltaico
GP2/GP3	Propósito General 2/3 (General Purpose 2/3)
gRPC	Llamada a procedimiento remoto de Google (Google Remote Procedure Call)
HCL	Lenguaje de Configuración de Hashicorp (HashiCorp Configuration Language)
HTML	Lenguaje de marcado de hipertexto (HyperText Markup Language)
HTTP	Protocolo de transferencia de hipertexto (HyperText Transfer Protocol)
IAM	Gestión de Identidad y Acceso (Identity and Access Management)
IDE	Entorno de Desarrollo Integrado (Integrated Development Environment)
IGW	Puerta de Enlace de Internet (Internet Gateway)
IPV4	Protocolo de Internet versión 4 (Internet Protocol version 4)
IPV6	Protocolo de Internet versión 6 (Internet Protocol version 6)
IVA	Impuesto sobre el Valor Añadido
JIT	Justo a Tiempo (Just InTime)
JSON	Notación de Objetos de JavaScript (JavaScript Object Notation)
KMS	Servicio de Gestión de Claves (Key Management Service)
LCU	Cantidad de unidades de capacidad del equilibrador de carga (Load Balancer Capacity Units)
LLVM	Máquina Virtual de Bajo Nivel (Low Level Virtual Machine)
MQTT	Transporte de Telemetría de Colas de Mensajes (Message Queuing Telemetry Transport)
NoSQL	No es solamente SQL (Not Only SQL)
ODS	Objetivos de Desarrollo Sostenible
ONU	Organización de las Naciones Unidas
PC	Computadora Personal (Personal Computer)
RAM	Memoria de acceso aleatorio (Random Access Memory)

REST	Transferencia Representacional de Estado (Representational State Transfer)
RPC	Llamada a procedimiento remoto (Remote Procedure Call)
S3	Servicio de Almacenamiento Simple (Simple Storage Service)
SQL	Lenguaje de Consulta Estructurado (Structured Query Language)
STOMP	Protocolo Simple de Mensajería Orientado a Texto (Simple (or Streaming) Text Oriented Messaging Protocol)
TLS	Seguridad de la capa de transporte (Transport Layer Security)
URL	Localizador uniforme de recursos (Uniform Resource Locator)
VPC	Nube Virtual Privada (Virtual Private Cloud)
VPN	Túnel Privado Virtual (Virtual Private Tunnel)
YAML	YAML no es un lenguaje de marcado (YAML Ain't Markup Language)

1 Introducción

En los últimos años, los avances en la computación escalable en la nube han habilitado nuevos modelos de trabajo, entre los cuales uno de los más destacados es la computación sin servidores, conocida en inglés como "**Serverless Computing**" [1].

Este modelo permite abstraer al usuario de la configuración, administración, gestión y escalamiento de los recursos de infraestructura, permitiéndole centrarse en el desarrollo y la ejecución de su aplicación, mientras que estas tareas son gestionadas por el proveedor del servicio [2].

Hoy en día, el modelo serverless es ampliamente utilizado por proveedores de servicios en la nube, quienes han desarrollado servicios bajo este modelo de negocio, enfocándose en la facturación por tiempo de ejecución o computación, conocido como FaaS (Function as a Service) [3]. Utilizando plataformas FaaS, un usuario puede ejecutar su aplicación en lenguajes como Java, Python o PHP, y solo se le factura por el tiempo de ejecución de su función de código, sin necesidad de preocuparse por las capas subyacentes.

Entre los proveedores destacados se encuentran AWS (Amazon Web Services) con su servicio Lambda, Google con Google Functions, y Microsoft con Azure Functions [4]. Al estudiar el estado del arte en cuanto a soluciones de código abierto y con licencias abiertas, se encuentran proyectos maduros que replican gran parte de la funcionalidad de los servicios mencionados, permitiendo a empresas o grupos ofrecer este servicio a su público objetivo, ya sean clientes o su propia comunidad de desarrolladores. Entre estas herramientas destacan Kubeless, OpenFaaS, Apache OpenWhisk y Knative [5].

En este proyecto de fin de grado se adaptará y modificará parte de la funcionalidad de un simulador fotovoltaico existente, llamado SISIFO, distribuyendo y adaptando un conjunto de sus funciones utilizando el lenguaje de programación Julia, y finalmente, convirtiendo la arquitectura monolítica existente en una arquitectura serverless mediante Knative y Kubernetes.

1.1 Marco y motivación del proyecto

El Grupo de Investigación en Sistemas Fotovoltaicos integrado en el Instituto de Energía Solar de la Universidad Politécnica de Madrid ha desarrollado y puesto a disposición del público el simulador SISIFO, gracias al soporte de la Comisión Europea a través de proyectos como PVCROPS [6] y MASLOWATEN [7].

Este simulador, de carácter abierto y con código disponible para cualquier usuario, implementa varios requisitos: el uso exclusivo de parámetros garantizados por fabricantes, la utilización de modelos validados según mediciones reales, y la precisión de dichos modelos, lo que permite reducir la incertidumbre de las estimaciones. Todo esto se basa en la experiencia desarrollada durante 20 años de medición, prueba y simulación de sistemas

fotovoltaicos (FV). Estos requisitos son garantías clave para la *financiabilidad* en última instancia.

SISIFO es una plataforma en desarrollo que recibe datos de diversos orígenes y utiliza tecnologías como MATLAB en sus capas internas, para procesar muchas variables que brindan al usuario una gran cantidad de información personalizable.

El uso de MATLAB es especialmente ventajoso [8] para el funcionamiento interno del simulador, ya que permite trabajar de manera óptima con operaciones matemáticas y ha sido históricamente considerada la herramienta "de facto" para todo lo relacionado con escalares, vectores y matrices, así como en numerosos escenarios de índole científico-técnica. Un ejemplo es el modelado de un panel fotovoltaico, como se aprecia en la figura 1.

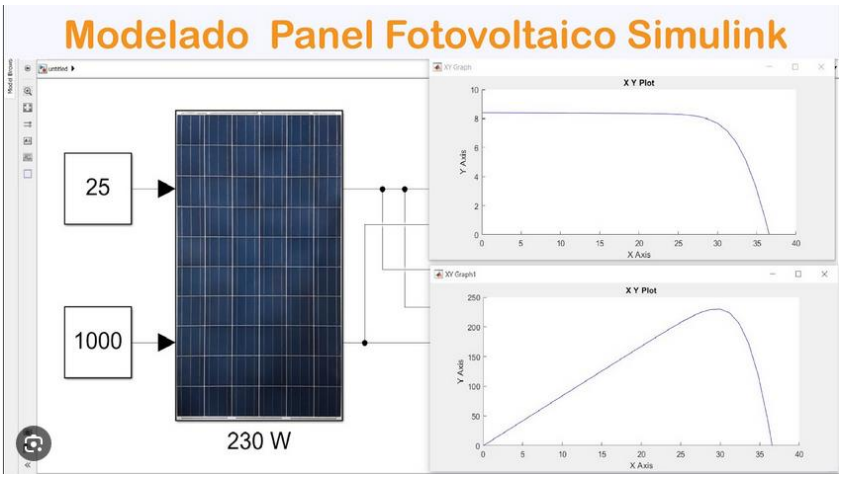
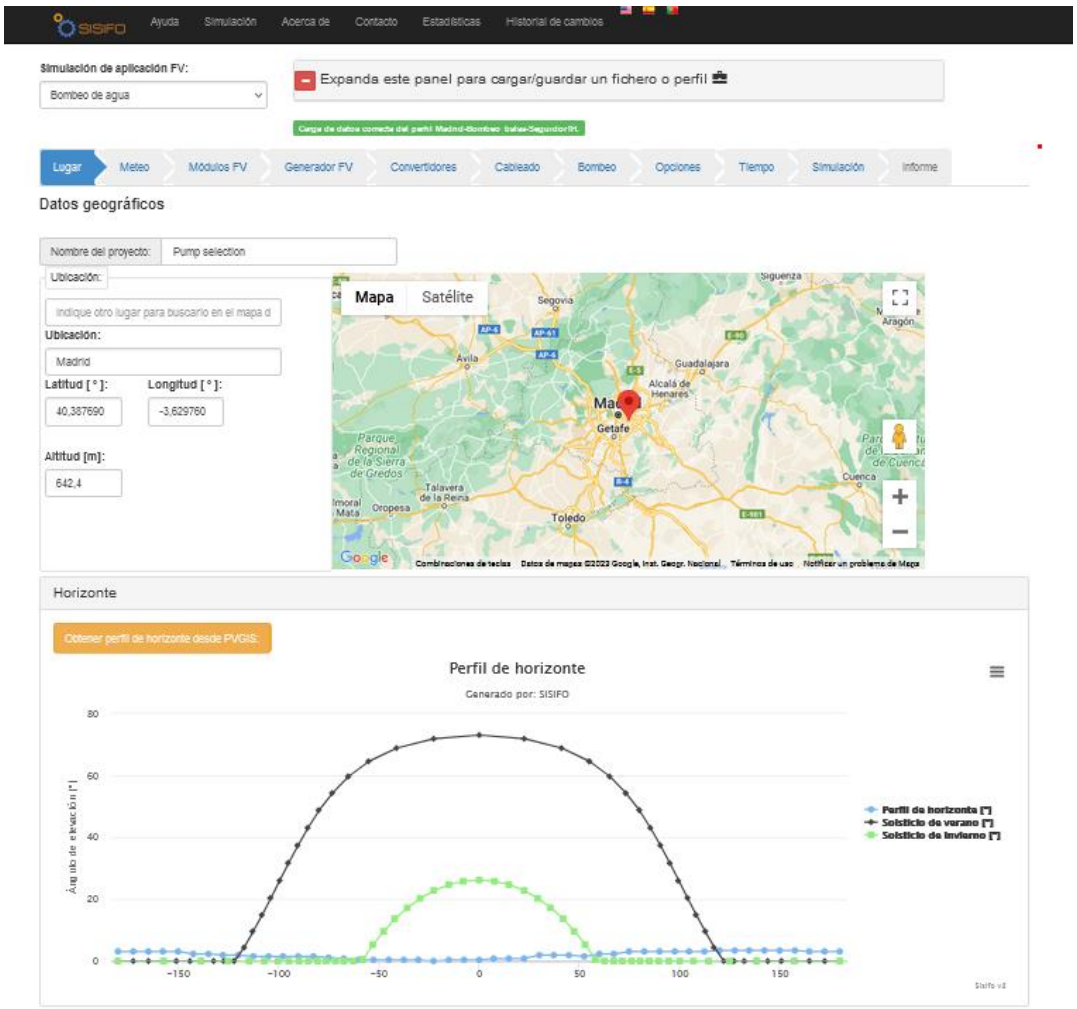


Figura 1. Ejemplo de uso de MATLAB *(Fuente: Matics Ingeniería Eléctrica)*

Sin embargo, cabe destacar que MATLAB presenta algunas desventajas. Su coste [9] de licencia y la necesidad de renovarla periódicamente suponen, en ocasiones, un precio prohibitivo para organizaciones que lo utilizan diariamente. Además, los fragmentos de código de gran tamaño y complejidad, en comparación con otros lenguajes como C++, se ejecutan a una velocidad significativamente más lenta, debido, en parte, a la naturaleza interpretada del lenguaje. Y aunque MATLAB es una potente herramienta para la modelización matemática o el análisis de datos, no todas las tareas de programación se benefician de sus cualidades.

En lo que respecta al funcionamiento interno de SISIFO, actualmente la arquitectura consta, a grandes rasgos, de un servicio que permite conectar con un *pool* de motores MATLAB.



Captura 2. Interfaz web de SISIFO *(Fuente: Elaboración propia)*

En la captura 2 es posible ver un ejemplo del funcionamiento de la interfaz web de SISIFO, accesible desde https://www.sisifo.info.

Gracias a este portal es posible recolectar una serie de parámetros, invocar la función deseada y presentar gráficamente los resultados. Esta invocación se realiza a una o más funciones monolíticas escritas en MATLAB.

1.2 Objetivos técnicos, académicos y de desarrollo

El objetivo del presente proyecto es la adaptación, el despliegue y el análisis de parte de la funcionalidad del simulador SISIFO sobre una arquitectura modelo *serverless*.

- La arquitectura actual, descrita en el apartado anterior, consta de funciones que, aunque pueden invocarse de forma concurrente, son implementaciones monolíticas con sus ventajas y desventajas. Estas últimas incluyen un código menos modular y más extenso; dificultad para actualizar, refactorizar o redistribuir el código; un alto nivel de dependencia entre los diferentes módulos del software; dificultad para escalar; y, en general, una falta de flexibilidad relativa. El diseño de MATLAB hace que no sea la herramienta más indicada para portar y adaptar el simulador a un modelo *serverless*. Se hace necesario explorar una alternativa que lo permita y de paso pueda suponer un aporte de ventajas en otras áreas.
- Por este motivo, se ha decidido usar Julia como lenguaje de programación, debido a su naturaleza multiplataforma, multiparadigma y específicamente orientada a la computación técnica y científica.
- Con el fin de evitar la vinculación directa con un proveedor de nube y crear un ecosistema portátil y no dependiente de tecnologías específicas, se ha escogido **Knative** como plataforma para implementar un modelo *serverless*.
- Finalmente, la infraestructura subyacente utilizará **Kubernetes** como orquestador de contenedores, que son las piezas atómicas sobre las que correrán las funciones de SISIFO.
- Todo esto busca ofrecer una posible alternativa para desplegar el simulador utilizando tecnologías de código abierto con fuerte soporte comunitario, explorando una arquitectura más modular, con un potencial superior y costos reducidos.

Desde el punto de vista académico, se pretende explorar y profundizar en las siguientes competencias y habilidades:

- **Entender las arquitecturas serverless:** Comprender las arquitecturas serverless, sus ventajas y limitaciones en comparación con las arquitecturas monolíticas tradicionales.
- **Mejorar las habilidades de programación:** Aprender y aplicar el lenguaje Julia en un contexto de computación técnica y científica.
- **Orquestación de contenedores:** Ganar experiencia práctica en la utilización de Kubernetes para la orquestación de contenedores.
- **Exploración de plataformas de código abierto:** Familiarizarse con tecnologías de código abierto, evaluando su soporte comunitario y viabilidad en proyectos de investigación y producción.
- **Desarrollo de arquitecturas modulares:** Aprender sobre la importancia y las prácticas de diseño de arquitecturas modulares y flexibles, orientadas al mantenimiento y escalabilidad.
- **Evaluación del rendimiento de una arquitectura:** Desarrollar la capacidad de analizar y evaluar el rendimiento de una arquitectura comparando diferentes modelos.

- **Consideración de costos en diseño de software:** Comprender el costo del desarrollo de software y tenerlo presente al tomar decisiones técnicas.
- **Desarrollo científico y técnico:** Aplicar de forma práctica los conocimientos técnicos aprendidos durante la carrera.
- **Investigación y documentación:** Desarrollar habilidades de investigación y capacidad para documentar y comunicar los hallazgos en un estilo académico.

Además, se pretende profundizar en las implicaciones e impacto de este proyecto en diferentes dimensiones, relacionando las acciones realizadas con los Objetivos de Desarrollo Sostenible marcados en la nueva Agenda 2030.

1.3 Estructura de la memoria

La estructura de la memoria del proyecto se compone de las siguientes secciones:

- **Marco tecnológico:** En esta sección se explica el estado actual de las tecnologías relacionadas con la computación "sin servidores". Se analizan diferentes proveedores y soluciones disponibles en el mercado, ofreciendo una visión general sin profundizar en los detalles específicos de cada tecnología.
- **Descripción de la solución propuesta:** Aquí se detalla el proceso de despliegue y construcción de la solución, con explicaciones paso a paso. Se incluyen descripciones detalladas en puntos clave para facilitar la comprensión del desarrollo del proyecto.
- **Resultados:** Esta sección presenta las pruebas realizadas tras la implementación de la solución, junto con los datos y resultados obtenidos.
- **Presupuesto:** Se proporciona un desglose del costo monetario total del proyecto.
- **Conclusiones:** Se concluye con un resumen de los datos e información recopilados durante el desarrollo del proyecto. Se analiza el impacto tecnológico, económico y ambiental del proyecto en la sociedad, así como su contribución a los Objetivos de Desarrollo Sostenible. Además, se proponen trabajos futuros para continuar y profundizar en las investigaciones realizadas.
- **Anexo:** Se incluye información adicional opcional sobre una de las herramientas utilizadas en las fases preliminares del proyecto.

Cada sección está diseñada para ofrecer una comprensión clara y completa del proyecto, desde la conceptualización hasta su impacto final.

2 Marco tecnológico

Definición de Serverless

El término *serverless* (sin servidor) comenzó a surgir a comienzos de la década de 2010. *Amazon Web Services* lanzó el primer servicio de computación en la nube *Lambda* en 2014 [10]. *Lambda* permitía que los desarrolladores ejecutaran código sin la necesidad de preocuparse por los servidores subyacentes, focalizándose únicamente en el código y la invocación de este (Figura 3). Este servicio llevó a la aparición de una nueva categoría de computación en la nube conocida como **FaaS** (*Functions as a Service*).

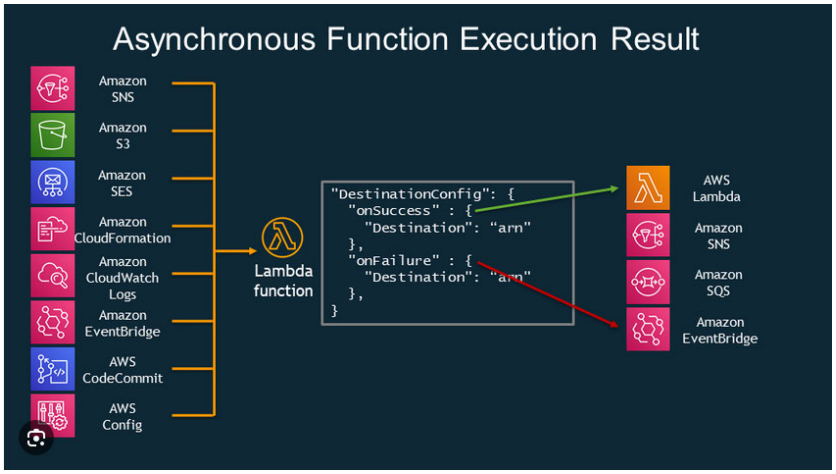


Figura 3. Ejemplo de integración entre servicios de AWS con Lambda *(Fuente: Amazon)*

Otros proveedores de servicios en la nube, como Microsoft con Azure Functions [11] y Google con Cloud Functions [12], incorporaron plataformas FaaS a su catálogo poco después. Hoy en día, la gran mayoría de los proveedores ofrecen esta funcionalidad.

Paralelamente, se desarrollaron herramientas de orquestación de sistemas para crear aplicaciones mucho más complejas utilizando esta metodología [13]. Aunque inicialmente la definición del término “serverless” se limitaba a aplicaciones ejecutadas en servicios de proveedores en la nube, ahora se entiende más bien como una solución impulsada por eventos y contenedores informáticos, utilizando recursos gestionados por el proveedor de la nube [14].

A pesar de que el término "sin servidor" sugiere la ausencia de recursos de infraestructura, en realidad lo que se abstrae al desarrollador es la configuración, gestión y mantenimiento de estos elementos, que son responsabilidad del proveedor de nube.

La adopción de la arquitectura serverless ha crecido rápidamente [3], ya que se ha demostrado que las aplicaciones basadas en este modelo pueden simplificar significativamente su desarrollo e implementación. Además, ayudan a reducir costos y mejoran la escalabilidad y flexibilidad.

Actualmente, muchas organizaciones líderes en todo el mundo utilizan [15] arquitecturas serverless para desarrollar y gestionar aplicaciones en línea, lo que demuestra que la evolución hacia tecnologías más distribuidas y simplificadas ha sido fundamental para satisfacer las cambiantes necesidades del mercado.

Ventajas

Esta arquitectura ofrece diversas ventajas tanto para los desarrolladores como para las organizaciones. Entre ellas se encuentran algunas como:

- **Escalabilidad automática:** El uso de esta arquitectura permite el no preocuparse por mantener servidores o infraestructura de apoyo. El desarrollador ejecuta su código y el proveedor de nube debe garantizar el servicio de forma “elástica” y de forma transparente.

- **Pago por uso:** Los recursos se asignan automáticamente a la aplicación y solo se cobra por el tiempo de ejecución de la aplicación y los recursos utilizados. Permite una mayor flexibilidad y ajuste en las cuentas.

- **Mayor agilidad:** El ahorro en tiempo se suma al de coste, ya que los plazos para ejecutar el código se reducen al no existir la necesidad de levantar y aprovisionar los servidores o piezas necesarias, tales como redes, balanceadores y otros.

- **Mejora de la eficiencia:** Los desarrolladores se centran en el desarrollo de la aplicación, sin preocuparse por la infraestructura subyacente, lo que aumenta la eficiencia y reduce los errores en el desarrollo (Figura 5).

- **Reducción de costos:** Al no necesitar invertir en infraestructura, una arquitectura modelo *serverless* puede reducir significativamente los costos de desarrollo y operación de aplicaciones. Además, elimina los costos asociados con la configuración y mantenimiento de servidores y recursos de infraestructura (Figura 4).

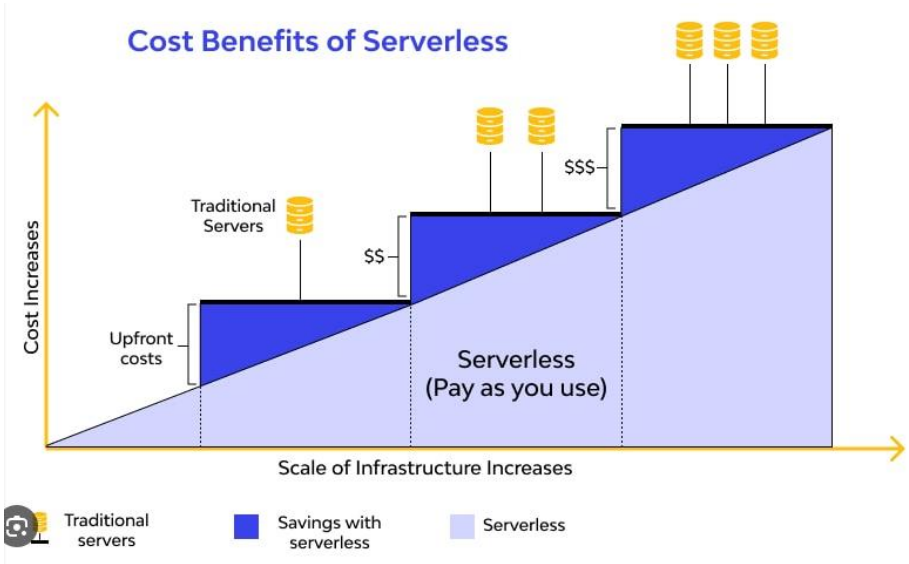


Figura 4. Relación de costes en serverless *(Fuente: Cloudflare)*

Desventajas

En la otra cara de la moneda, se pueden encontrar algunos inconvenientes a la hora de implementar las aplicaciones con este modelo:

- **Tiempo de puesta en marcha:** Puede haber un pequeño retraso en el tiempo de arranque del servicio debido a que los contenedores subyacentes se están iniciando y preparando para manejar la carga. Esto puede llegar a ser un problema en aplicaciones que requieren una alta capacidad de respuesta.

- **Conectividad:** Las funciones *serverless* generalmente se conectan a otros servicios o sistemas para obtener o proporcionar información. A veces, la conexión a otros recursos puede ser compleja y este tipo de arquitectura puede no ser la opción más sencilla.

- **Limitaciones de recursos:** Al depender de los recursos asignados por el proveedor, las limitaciones de alojamiento y recursos pueden estar presentes. Si la demanda supera la capacidad provista por el proveedor, la aplicación puede encontrar problemas de rendimiento.

- **Complejidad en el testeo:** Debido a la complejidad de la arquitectura *serverless*, puede ser más difícil probar y depurar aplicaciones en comparación con las aplicaciones tradicionales, lo que puede llevar a problemas de calidad.

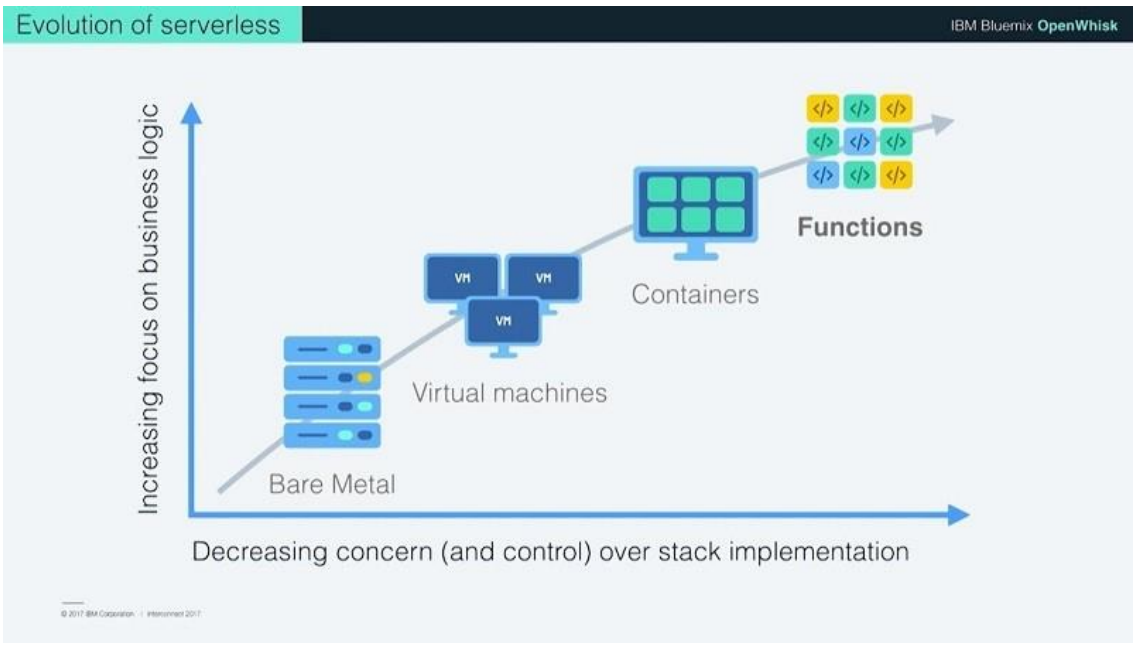


Figura 5. Relación lógica de negocio / preocupación y control del *stack (Fuente: Nubity)*

Diferencias con otros modelos

De las principales arquitecturas escogidas actualmente para el desarrollo de aplicaciones se suelen destacar tres [16]:

- **Monolitos:** Arquitectura básica, en la que la aplicación es una implementación única conteniendo todo el código para funcionar. Normalmente este código suele residir en un solo repositorio. Es un modelo tradicional que, aunque hoy en día tiene algunos detractores, sigue siendo perfectamente válido. Entre sus ventajas se tiene un desarrollo y despliegues rápidos, sencillos y económicos; y en contraparte se tiene una escalabilidad menor, falta de adaptabilidad, y una mayor probabilidad de "código espagueti", término utilizado para referirse a la complejidad y no modularidad del código.

- **Microservicios:** Arquitectura en la que una aplicación se estructura como una colección de diversos servicios, cada uno orientado a una funcionalidad única, interconectados entre sí para completar una lógica final. No necesariamente se hace referencia a "micro" como "pequeño", sino más bien a una unidad o separación lógica dentro del conjunto global. Las buenas prácticas de esta arquitectura dictan que un microservicio se implementa y tiene un ciclo de vida independiente de los demás, no suelen compartir persistencia (como bases de datos), y son escalables tanto horizontal como verticalmente sin afectar al resto de la aplicación. Las ventajas de usar esta arquitectura son, por ejemplo, una mayor flexibilidad a la hora de escoger tecnologías o lenguajes de implementación; una mayor escalabilidad; un despliegue rápido; y una

mayor modularidad e independencia. Por el contrario, aumenta la complejidad y la posibilidad de fallas en el sistema final, especialmente en organizaciones no tan maduras; y supone mayor latencia por la comunicación entre piezas.

En la figura 6 es posible apreciar la diferencia entre varios modelos de arquitectura y un ejemplo de la facilidad para escalar si la demanda se incrementa.

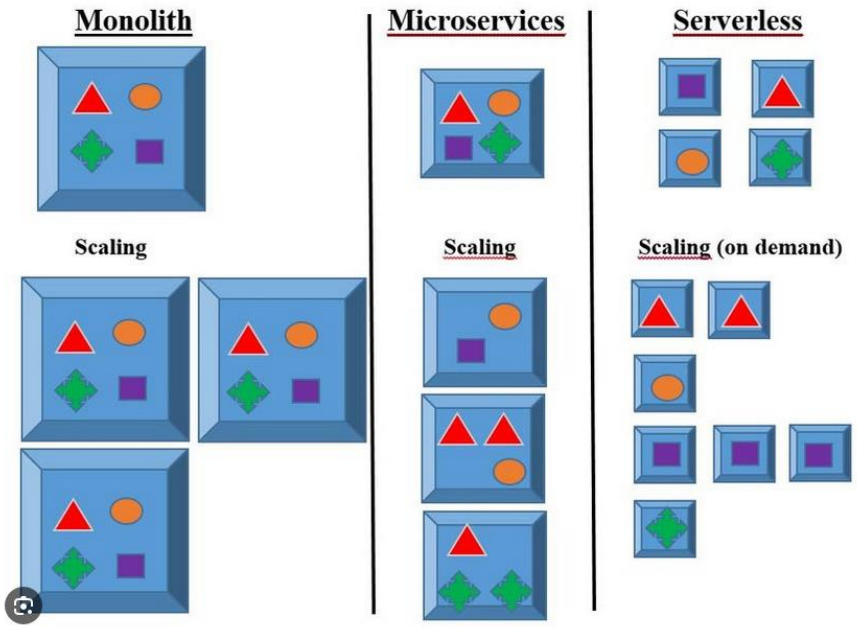


Figura 6. Comparativa entre modelos de arquitectura *(Fuente: Alex Kaplunovich* [17]*)*

2.1 Implementación de serverless

Como se ha indicado previamente, la mayoría de los proveedores de nube tienen servicios FaaS en los que ejecutar código bajo demanda. Entre los más destacados se pueden encontrar los listados a continuación.

2.1.1 AWS Lambda

Figura 7. Logotipo de AWS Lambda *(Fuente: Amazon)*

AWS Lambda (Figura 7) es un servicio ofrecido por Amazon Web Services lanzado en 2014 que permite a los desarrolladores ejecutar código en respuesta a eventos o solicitudes, sin necesidad de preocuparse por administrar infraestructura.

Es compatible con varios lenguajes de programación, incluyendo Python, Java, JavaScript, TypeScript, Ruby, Go y C#.

Ofrece diversas características, como la facilidad de integración con otros servicios de AWS, la escalabilidad automática de aplicaciones, la alta disponibilidad, la precisión en la facturación y la seguridad mejorada a través de la implementación de múltiples capas de aislamiento de recursos [18].

A diferencia de otros servicios de AWS, los cuales cobran por hora, pero miden en segundos, AWS Lambda mide en picos de 100 milisegundos, por lo que ofrecen un servicio de pago bajo demanda muy exacto.

2.1.2 Otros FaaS en la nube

No solo ofrece esta funcionalidad Amazon. Empresas como Microsoft, Google o IBM disponen, entre sus servicios de computación, plataformas para ofrecer funciones como servicio a sus clientes [19].

En la figura 8 se ofrece una pequeña comparativa con las diferencias entre proveedores, siempre cambiantes y en constante evolución.

Entre estas características suelen encontrarse los lenguajes de programación admitidos para codificar las funciones, el número de ejecuciones concurrentes; o incluso la forma de desplegar los paquetes de código (Por ejemplo, cargando el código desde un fichero comprimido, o utilizando el existente de un almacenamiento de objetos en la nube)

Features	Amazon Web Services (AWS)	Microsoft	Google	IBM
Serverless Compute Offerings	Lambda	Azure Functions	Google Cloud Functions	IBM Cloud Functions
Maximum Functions	Unlimited	Unlimited	1,000 per project	Unlimited
Scalability & Availability	Transparent – Automatic Scaling	Automatic Scaling	Automatic	Automatic
Languages Supported	Node.js, Python, Java, C#, Go, Ruby	C#, JavaScript, F#, Python, TypeScript	Node.js, Python, Go	Node.js, Go, Swift, PHP, Java, .NET, Python, Ruby
Concurrent Executions	1,000 concurrent executions per account per region	10 concurrent executions per function	400 per function	1,000 per project
Deployments	.ZIP to S3 or Lambda	GitHub, Visual Studio, Local git, Dropbox, Bitbucket	Google Cloud Source, .ZIP to cloud storage	GitHub, Bluemix DevOps
Concurrent Executions	1,000 concurrent executions per account per region	10 concurrent executions per function	400 per function	1,000 per project
Code Size	50MB compressed 250MB uncompressed	None, user pays the storage cost	100MB compressed 500MB uncompressed	48MB

Figura 8. Diferencias entre plataformas FaaS *(Fuente: PanonIT)*

2.1.3 Alternativas *on premises*

A menudo, cuando se habla de "serverless", se asocia este término principalmente con entornos en la nube. Sin embargo, es factible implementar plataformas serverless en ambientes *on-premises*, como centros de procesamiento de datos (CPD) locales o servidores físicos dedicados. No obstante, esta práctica no es tan común, ya que el enfoque serverless está diseñado para ejecutar funciones en respuesta a eventos específicos.

La mayoría de los proveedores de servicios en la nube ofrecen soluciones y servicios integrados que facilitan la generación de estos eventos y disparadores. Por ejemplo, en el caso de AWS, un usuario puede almacenar archivos JSON en Amazon S3, y gracias a la integración nativa de AWS Lambda con S3, es posible detectar y procesar estos archivos de forma casi automática.

A pesar de las ventajas de las soluciones en la nube, una organización podría optar por desarrollar su propia implementación o plataforma serverless, motivada por razones de seguridad, cumplimiento normativo, protección de datos, la innovación en funcionalidades de

la plataforma FaaS, evitar la dependencia de proveedores externos, o la necesidad de operar en un entorno *multicloud* [20].

Hoy en día, existen *frameworks* de código abierto que soportan múltiples lenguajes de programación y se integran con tecnologías como el empaquetado de aplicaciones en contenedores, virtualización, bases de datos NoSQL, y otros servicios típicos de proveedores de nube. Algunas de estas soluciones se describen en siguientes apartados.

2.1.4 OpenFaaS

OpenFaaS (Open Functions as a Service) es una plataforma serverless de código abierto diseñada para facilitar a los desarrolladores la ejecución de funciones sin servidor de manera rápida y sencilla. Esta plataforma se basa en contenedores y utiliza Docker para crear y ejecutar funciones dentro de estos contenedores [21].

Proporciona una capa de abstracción adicional sobre la orquestación de contenedores y la gestión de recursos. Una de sus principales ventajas es la compatibilidad con múltiples lenguajes de programación, como Node.js, Python, PHP, Ruby y Java, lo que permite a los desarrolladores elegir el lenguaje que mejor se adapte a cada aplicación y función.

Ofrece, además, características avanzadas de seguridad, como sistemas de autenticación y autorización. Se integra con sistemas de monitoreo y alerta, lo que ayuda a los desarrolladores a supervisar el rendimiento y la salud de las funciones implementadas.

OpenFaaS es flexible en su implementación, ya que puede utilizarse en diferentes plataformas de nube, incluidas Amazon Web Services, Microsoft Azure y Google Cloud Platform, así como en infraestructuras locales o sobre Kubernetes, ofreciendo versatilidad tanto en entornos cloud como on-premises [22].

2.1.5 Apache OpenWhisk

Es una plataforma de computación en la nube de tipo Function as a Service (FaaS) de código abierto que permite a los desarrolladores crear y ejecutar código en respuesta a eventos, como solicitudes de API, cambios en los datos, cargas de trabajo programadas, entre otros. Originalmente creado por IBM, este proyecto lo mantiene la fundación Apache [23].

Está diseñado para ser un entorno de ejecución de funciones distribuido y escalable, lo que lo hace adecuado para aplicaciones de baja latencia y alta disponibilidad (Figura 9).

OpenWhisk en su mayor parte está escrito en Scala (Alrededor de un 80% según las estadísticas de su repositorio público), y lleva más de cinco años en activo, lo que lo hace especialmente estable, si bien no tiene un ciclo de desarrollo muy continuado [24].

El proyecto cuenta con un API REST y una herramienta de línea de comandos, soporte para empaquetado, catálogo de servicios y posibilidad de despliegue en multitud de opciones de empaquetado de código en contenedores [25].

OpenWhisk se compone de una arquitectura modular y permite la integración con servicios externos. Además, está disponible en múltiples proveedores de nube, o en infraestructura local.

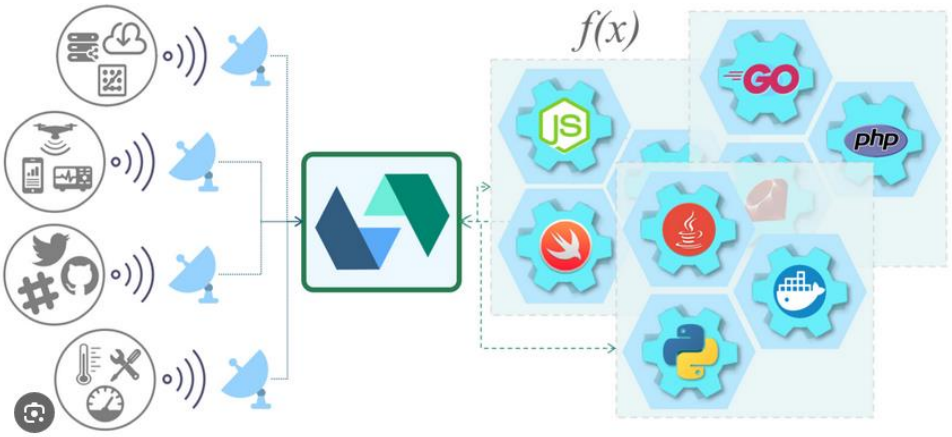


Figura 9. Ejemplo de integración de OpenWhisk *(Fuente: Ravi Nemala)*

2.1.6 Knative

Knative es un proyecto de código abierto impulsado por Google [26], que se enfoca en hacer que sea más fácil construir y operar aplicaciones modernas en la nube en un entorno de Kubernetes. Knative proporciona una plataforma *serverless* desplegada sobre Kubernetes.

Knative permite a los desarrolladores crear funciones y eventos completamente administrados y automáticos utilizando APIs de alto nivel. Aporta una plataforma de eventos con funciones como ejecución de contenedores, escalado automático, configuración de rutas y gestión de revisiones, lo que permite a los equipos de desarrollo enfocarse en escribir código de calidad y no preocuparse por la operación de la infraestructura subyacente [27].

Knative es altamente flexible y compatible con cualquier plataforma de nube que soporte Kubernetes, lo que permite su uso en varias plataformas de nube pública o distribuciones de Kubernetes privadas, incluyendo despliegues *on-premises*. La relación de actores principales se muestra en la figura 10.

Figura 10. Relación entre actores en Knative *(Fuente: Aquasec.com)*

2.2 Contenedores y orquestadores

Estas tecnologías no están destinadas únicamente al uso de funciones como servicio, pero son habituales en estos escenarios:

2.2.1 Docker

Docker es una plataforma de software que permite a los desarrolladores empaquetar, distribuir y ejecutar aplicaciones en contenedores. Estos contenedores incluyen todas las dependencias y configuraciones en una única unidad estándar. En la figura 11 se presenta una comparación entre una máquina virtual y un contenedor.

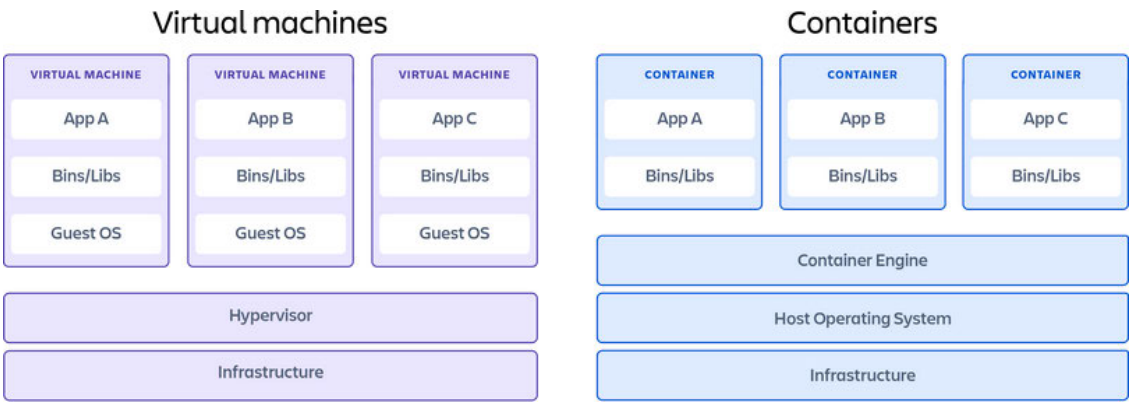


Figura 11. Diferencia entre contenedor y máquina virtual *(Fuente: Atlassian)*

Docker simplifica el proceso de creación de entornos de desarrollo y producción, ya que los contenedores son portátiles y se pueden mover fácilmente entre diferentes sistemas operativos y plataformas de nube. Además, los contenedores se pueden escalar y actualizar de forma independiente, lo que mejora la eficiencia de la administración de infraestructura.

Los contenedores también permiten a los desarrolladores trabajar con diferentes versiones de lenguajes de programación y bibliotecas sin problemas de compatibilidad.

Es importante diferenciar las diferentes partes de un sistema de contenedores (Figura 12). Normalmente se suele usar Docker para referirse a todas las piezas del *stack* que incluyen el motor, el *runtime* e incluso el cliente para operar con el repositorio de imágenes de contenedores [28].

Hoy se puede operar con diferentes proyectos o herramientas para cada pieza. Por ejemplo, es posible usar *containerd* como *runtime* de contenedores, y seguir instanciando contenedores a partir de imágenes construidas con Docker y alojadas en su repositorio (En inglés, *registry*) oficial.

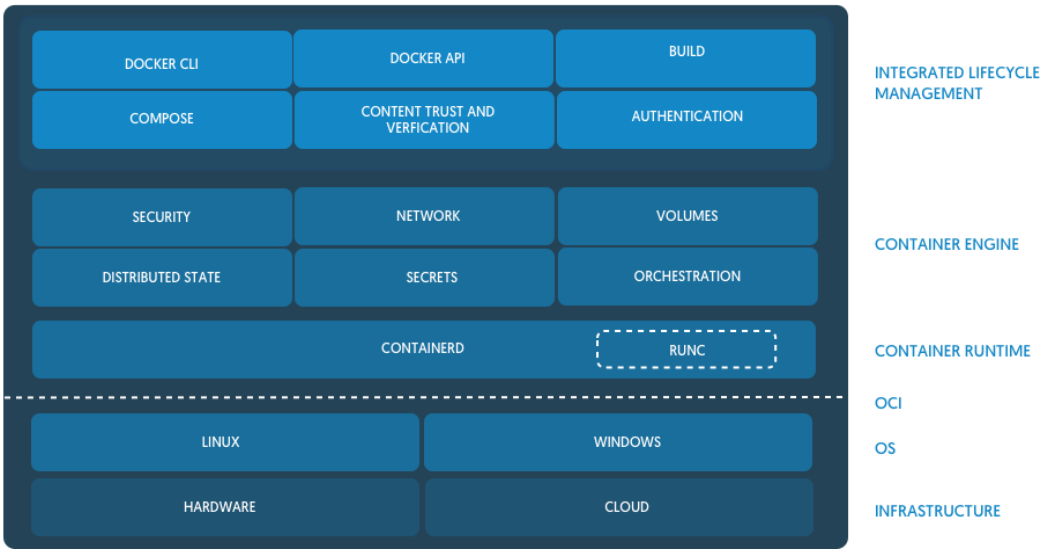


Figura 12. Diferentes partes de un sistema de contenedores *(Fuente: Tanju Ravi Tao)*

2.2.2 Kubernetes

Kubernetes es una plataforma de orquestación de contenedores de código abierto desarrollada originalmente por Google [29]. Permite automatizar la implementación, escalado y administración de aplicaciones ejecutadas en contenedores.

Kubernetes gestiona los recursos de cómputo, almacenamiento y redes, y permite a los desarrolladores realizar la implementación de forma declarativa, normalmente mediante ficheros YAML, especificando el estado deseado de la aplicación y sus componentes. El

orquestador se encarga automáticamente de las actividades de implementación y actualización para garantizar que la aplicación esté siempre en el estado deseado.

Kubernetes se puede utilizar con diferentes *runtimes* de contenedores, incluyendo Docker, containerd y otros; y se integra con diferentes herramientas de automatización, como Ansible, Chef y Puppet. Además, se puede utilizar en diferentes plataformas de nube, como Amazon Web Services, Microsoft Azure y Google Cloud Platform, así como *on-premise* [30].

Se muestra un ejemplo de arquitectura básica de Kubernetes en la figura 13:

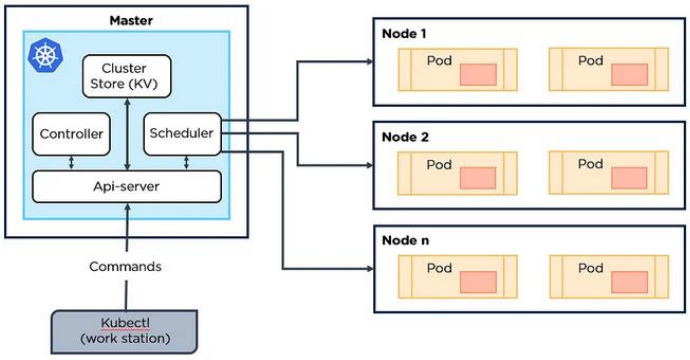


Figura 13- Arquitectura de un clúster de Kubernetes *(Fuente: Nomzy-Kush, Medium)*

2.3 Lenguajes de implementación

En el análisis de alternativas a la hora de implementar funciones destinadas a ejecutarse sobre la arquitectura propuesta, los principales lenguajes de programación identificados aptos para el funcionamiento del simulador SISIFO son los siguientes:

2.3.1 MATLAB

MATLAB es una plataforma de programación orientada a ingenieros y científicos creada por Mathworks. Se usa para analizar, diseñar e implementar sistemas y productos de índole técnica [31].

MATLAB se caracteriza por tener un lenguaje propio [32], basado en matrices, muy orientado al cálculo; incluyendo vectores, matrices, escalares y otros elementos matemáticos. No obstante, es posible utilizarlo para programación orientada a objetos. En cualquier caso, se considera que su punto fuerte es la facilitación de expresiones de carácter matemático en ambientes computacionales.

Tiene un IDE integrado y representación gráfica de modelos 2D y 3D.

Su funcionalidad puede extenderse mediante *toolboxes*. Estas "cajas de herramientas" son paquetes adicionales que permiten añadir nuevas vías de trabajo e incluso abarcar nuevas áreas tecnológicas y de ingeniería.

MATLAB cuenta con numerosas ventajas [33]:

- Relativamente fácil de usar.
- Interfaz gráfica.
- Gran comunidad y soporte de usuarios.
- Funcionalidad extensible.
- Disponible para la mayoría de las plataformas (Windows, Mac, UNIX).

Pero a su vez se pueden citar dos grandes desventajas:

- Coste de licencia (Algo que para entornos educativos puede no ser del todo un problema por el modelo de negocio de Mathworks [9])
- Lenguaje interpretado y baja velocidad de ejecución.

2.3.2 Julia

Julia es un lenguaje de programación gratuito y de código abierto creado en 2009 y puesto a disposición de los usuarios en 2012 por A. Edelman, S. Karpinski, J. Bezanson y V. Shah [34].

Se describe como un lenguaje *homoicónico* (esto es, la representación primaria de este lenguaje es también una estructura de datos en un tipo primitivo), multiplataforma y multiparadigma. Es de tipado dinámico y está orientado a la computación técnica y científica (aunque puede ser usado con carácter general). Consta de un compilador JIT (Just-In-Time) [35].

Entre sus ventajas se tiene que:

- Está orientado a ser de alto rendimiento.
- Tiene un gran enfoque en el análisis numérico.
- Se puede elegir entre diferentes patrones de programación según las necesidades.
- Es dinámico, por lo que puede usarse de forma interactiva.
- Tiene sintaxis de alto nivel y es considerada fácil de aprender. Se suele comparar con Python.
- Tiene una biblioteca estándar extendida y multitud de paquetes de terceros.

Julia ha ganado popularidad especialmente en el ámbito académico y en la industria donde el alto rendimiento computacional es crucial, debido a su capacidad de combinar la velocidad de lenguajes como C o Fortran con la facilidad de uso de lenguajes interpretados como Python [36]. Su compilador JIT permite ejecutar código prácticamente a la velocidad del código nativo, lo que lo hace ideal para aplicaciones que requieren cálculos intensivos y modelado matemático complejo [37].

Además, Julia dispone de JuliaHub [38], una plataforma que facilita el desarrollo colaborativo y la implementación de proyectos complejos. Su interoperabilidad con otros lenguajes, como Python o R [39], eleva aún más su flexibilidad en ambientes multidisciplinarios. La comunidad de Julia sigue creciendo, impulsando desarrollos como herramientas para machine learning, optimización matemática y simulaciones en tiempo real [40].

2.3.3 Python

Python, creado por Guido van Rossum a finales de los años ochenta [41], y mantenido hoy en día por la Python Software Foundation, es un lenguaje de programación interpretado de alto nivel utilizado en desarrollo de software, aplicaciones web, ciencia de datos y *machine learning*, entre otros muchos ámbitos.

Se considera eficiente y fácil de aprender, ya que en su filosofía [42] se hace hincapié en la legibilidad y claridad del código. Es gratuito, de código abierto y multiplataforma. Es un lenguaje multiparadigma y soporta en parte la orientación a objetos, la programación imperativa, e incluso la programación funcional.

En 2024 Python sigue siendo uno de los lenguajes de programación más populares [43]. La larga trayectoria de Python es sinónimo de que es un lenguaje que seguirá existiendo en los próximos años. Su extensibilidad mediante miles [44] de paquetes es clave de su popularidad, haciéndolo totalmente multipropósito.

De cara al uso en análisis numérico y científico, Python ofrece una amplia variedad de bibliotecas y herramientas que facilitan el manejo y procesamiento de datos complejos. Entre las más destacadas se encuentran NumPy [45] para la manipulación de arrays y operaciones matemáticas eficientes, Pandas para el manejo de datos estructurados a través de dataframes, SciPy [46] que amplía las capacidades de álgebra y optimización, y Matplotlib para la creación de visualizaciones y gráficos de alta calidad. Además, en el ámbito del machine learning, Python se beneficia de un ecosistema robusto con librerías como scikit-learn para modelos tradicionales, y TensorFlow [47] y PyTorch para el desarrollo de redes neuronales y modelos de aprendizaje profundo. Esta combinación de capacidades científicas y de machine learning posiciona a Python como una herramienta esencial en la investigación académica y el desarrollo profesional [48].

Su comunidad activa y extensa asegura un soporte constante y la actualización continua de estos recursos, lo que refuerza su posición como un lenguaje preferido tanto para principiantes por su claridad y simplicidad, como para expertos que buscan una plataforma potente y versátil que satisface diversas necesidades en la industria de la tecnología.

2.4 Tecnologías auxiliares

En este apartado se detallan herramientas y tecnologías exploradas a lo largo del proyecto y que son de utilidad para el desarrollo de este, complementarias a las descritas previamente.

2.4.1 Terraform

Terraform es una herramienta de infraestructura como código (IaC). A través de la definición de la infraestructura en archivos de configuración haciendo uso de un lenguaje específico de dominio, HCL (HashiCorp Configuration Language), y de forma declarativa, Terraform permite construir, modificar y versionar la infraestructura de manera segura y coherente. Este enfoque reduce significativamente los errores manuales y asegura que los entornos de desarrollo y producción se mantengan consistentes [49].

En el núcleo de Terraform se encuentran los "*providers*" o proveedores, elementos fundamentales que actúan como interfaces hacia los diferentes servicios. Los proveedores permiten a Terraform interactuar con una amplia gama de servicios, desde plataformas en la nube como AWS, Azure o Google Cloud, hasta recursos *on-premises*. Esta funcionalidad permite abstraer y simplificar la comunicación con las API y particularidades de estos servicios (Figura 14).

Asimismo, los módulos de Terraform representan un nivel avanzado de organización y reutilización de configuraciones de infraestructura [50]. Un módulo puede definirse como un conjunto de componentes de infraestructura agrupados para cumplir con un objetivo funcional específico, como el despliegue de una arquitectura de red completa. Este enfoque modular no solo mejora la eficiencia al permitir la reutilización de configuraciones previamente definidas, sino que también promueve mejores prácticas y estandarización dentro de los equipos.

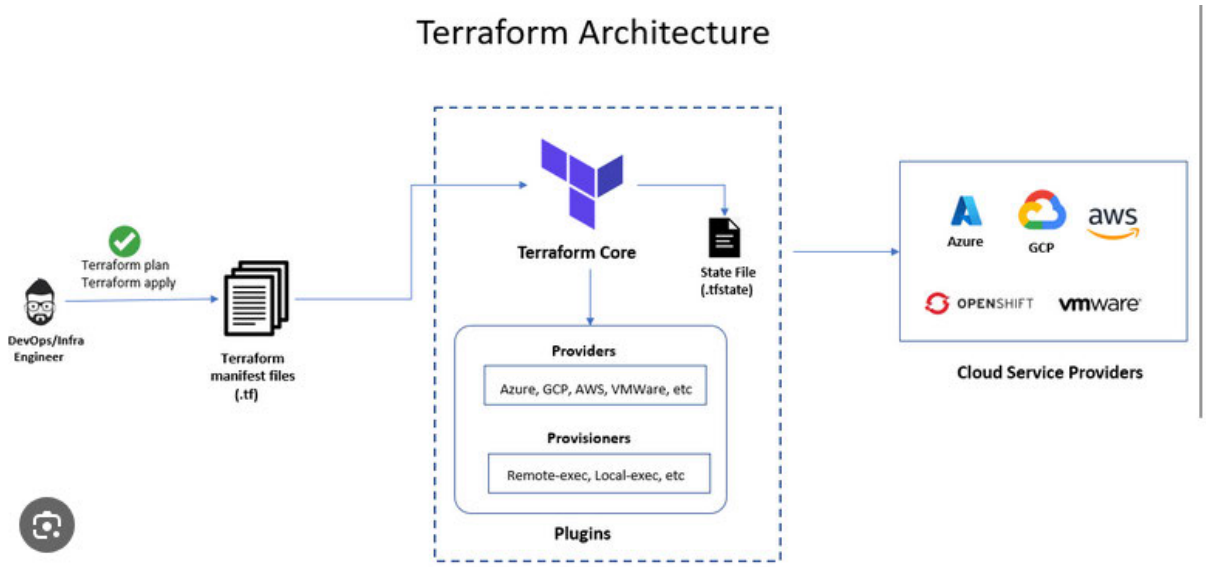


Figura 14. Funcionamiento de Terraform *(Fuente: Vivekanand Rapaka)*

2.4.2 RabbitMQ

RabbitMQ es un sistema de mensajería avanzado enmarcado dentro de la categoría de *brokers* de mensajes de código abierto. Un broker de mensajes es un componente clave que permite que las diferentes partes de una aplicación intercambien mensajes de manera asincrónica. En esencia, actúa como un intermediario para el intercambio de datos entre aplicaciones distribuidas.

En el ámbito de su uso cotidiano, RabbitMQ destaca por implementar patrones de mensajería que son esenciales para operaciones complejas, entre ellas el procesamiento de tareas en segundo plano, la integración entre distintos servicios y la gestión de colas de mensajes. Soporta múltiples protocolos de mensajería, incluyendo AMQP, STOMP y MQTT [51].

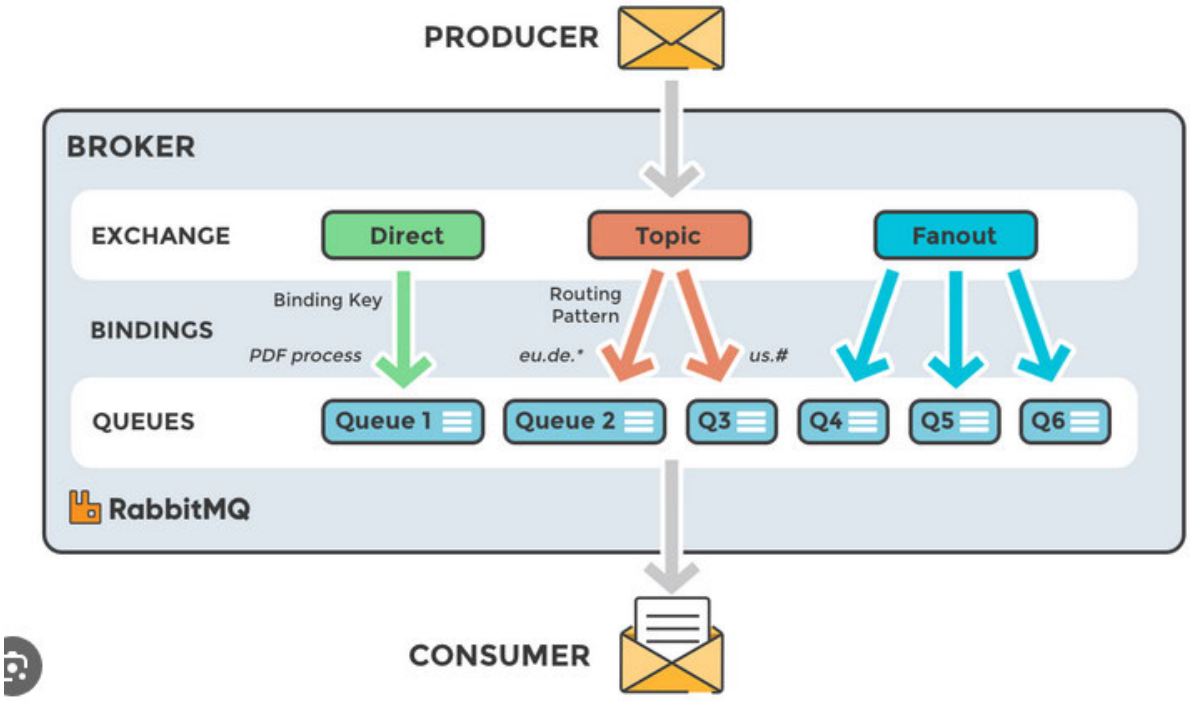


Figura 15. Arquitectura de RabbitMQ *(Fuente: CloudAMPQ)*

La arquitectura de RabbitMQ (Figura 15) involucra conceptos como colas, intercambios y *bindings*, que juntos forman un sistema de enrutamiento de mensajes. Los mensajes son publicados por productores en intercambios, los cuales deciden qué colas reciben esos mensajes según reglas de enrutamiento predefinidas. Los consumidores recuperan mensajes de estas colas.

2.4.3 Memcached

Memcached es una solución de almacenamiento en caché que se emplea principalmente para acelerar aplicaciones web al reducir la carga en las bases de datos. Se trata de un sistema de memoria distribuida que almacena datos y objetos en RAM, proporcionando acceso rápido a información temporal solicitada con frecuencia, algo especialmente valioso en entornos donde el tiempo de respuesta y el rendimiento son críticos.

Actúa como un almacén de datos en memoria. Cuando una aplicación necesita acceder a datos que ya han sido procesados anteriormente, puede recuperar esos datos directamente desde la memoria en lugar de ejecutar nuevamente consultas lentas en la base de datos o procesar datos de entrada.

Tiene un diseño distribuido (Figura 16), liviano y ágil [52], basado en un modelo cliente-servidor donde los datos se almacenan como pares clave-valor, que permite una fácil integración en diversas arquitecturas de software.

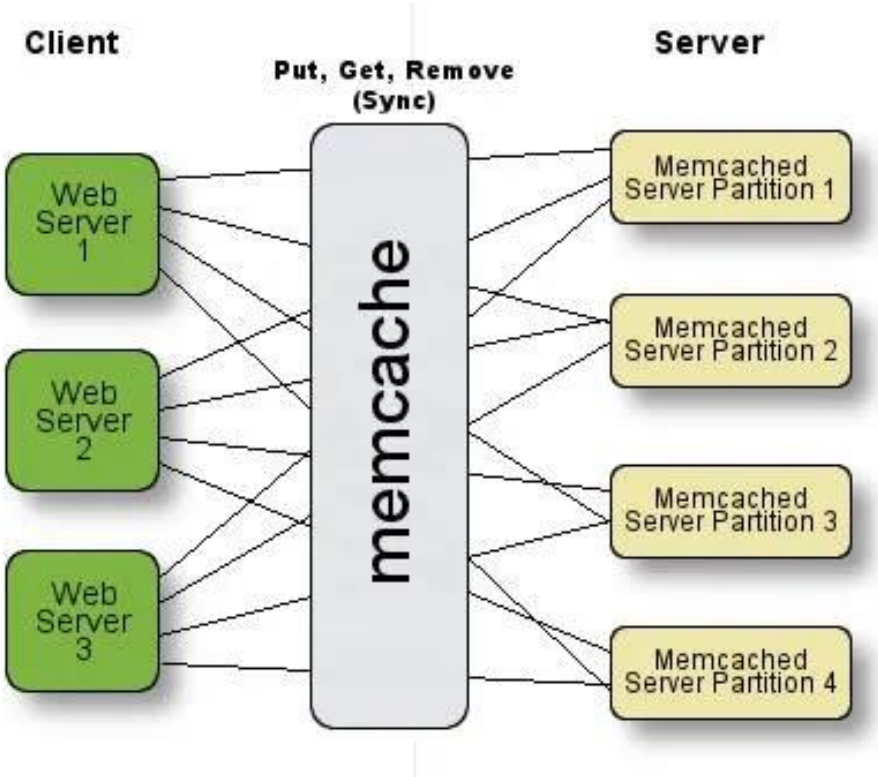


Figura 16. Arquitectura distribuida de Memcache *(Fuente: Renjith Raju)*

3 Descripción de la solución

La solución propuesta es la reescritura a Julia y la adaptación a una arquitectura basada en microservicios de un fragmento de una primera versión de SISIFO escrita en PHP y disponible públicamente.

Para ello el desarrollo se ha separado en fases diferenciadas explicadas con mayor detalle en cada punto del capítulo. A nivel global estas son las acciones realizadas:

- Se ha desarrollado una solución de arquitectura en la cual se ha desplegado un clúster de Kubernetes en la nube de Amazon Web Services (AWS) [53], utilizando Terraform [50] como herramienta de infraestructura como código (IaC) para automatizar el despliegue de todas las piezas necesarias. Esta aproximación permite una gestión eficiente, declarativa y reproducible. En un segundo nivel, el uso de Kubernetes proporciona una plataforma robusta y escalable para manejar contenedores, donde se ejecutan los microservicios.
- Una vez configurado el clúster, se ha procedido a instalar Knative haciendo uso de su operador. Knative extiende las capacidades de Kubernetes y su API al ofrecer funciones adicionales para la gestión de aplicaciones *serverless* y eventos. Se hace uso de uno de los dos grandes componentes de Knative: *serving* [54], descrito más adelante.
- Posteriormente, el enfoque se pone en el proceso de migración de funciones específicas del simulador fotovoltaico SISIFO. Se seleccionan los fragmentos de código más relevantes para el proyecto, se realiza un estudio del código existente, y se hace un trabajo de migración y adaptación.
- Por último, se garantiza que las nuevas funciones de código se empaqueten y ejecuten como microservicios independientes, y que estos estén interconectados eficientemente haciendo uso de Knative, habilitando la comunicación y el intercambio de datos requerido para el correcto funcionamiento del fragmento del simulador tratado.

3.1 Infraestructura base

El despliegue de la infraestructura que sustenta el resto de los componentes del proyecto, se apoya en la herramienta Hashicorp Terraform.

Se hace uso de módulos creados por la comunidad y de acceso público y libre para el despliegue de las piezas clave, como sería el caso de una librería pública en un lenguaje de programación. Es posible además especificar variables para parametrizar su comportamiento.

Estos módulos se orquestan como bloques en un *stack* final y su relación de dependencias se gestiona automáticamente por la herramienta.

El diagrama de la arquitectura propuesta plasmado en la figura 17 (Se simplifica a dos zonas de disponibilidad de las tres desplegadas por claridad del diagrama) muestra el resultado final del uso del *stack* compuesto de módulos creado para este proyecto:

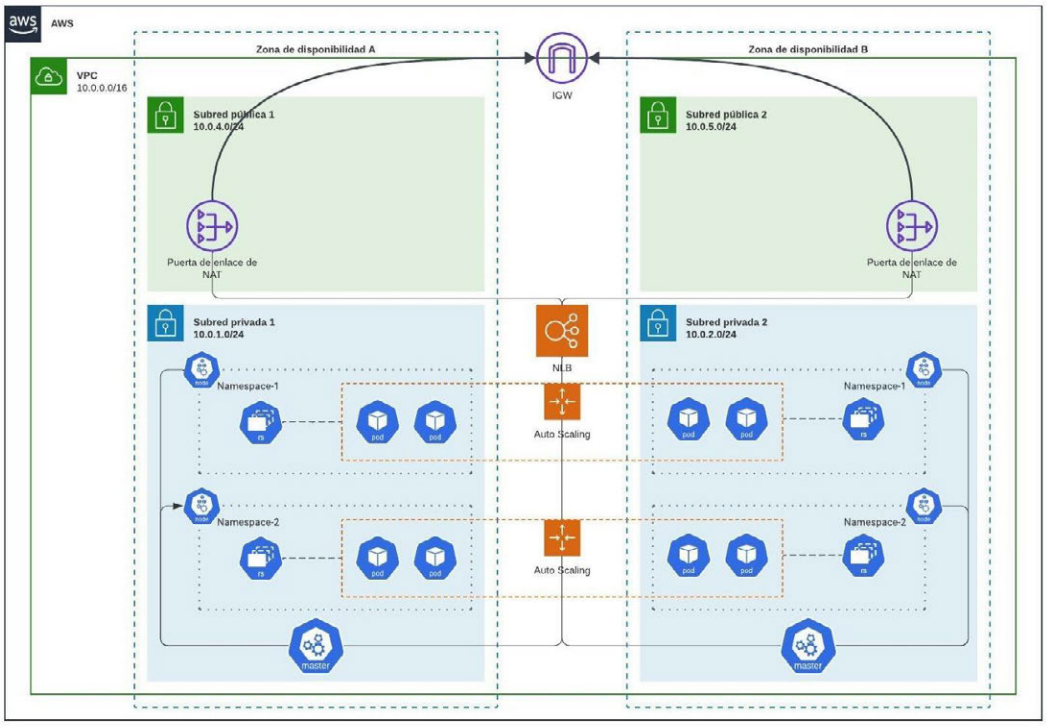


Figura 17. Arquitectura clúster Kubernetes en AWS *(Fuente: Elaboración propia)*

El despliegue de la arquitectura de la figura anterior requiere de 67 recursos de AWS, contando elementos de red, componentes computación y recursos complementarios.

La mayoría de ellos se han configurado y personalizado explícitamente como un fichero de definición de infraestructura haciendo uso del *provider* de Terraform para AWS. El resultado es un *stack* llamado k8s-knative subido a GitHub a un repositorio, para poder ser reutilizado.

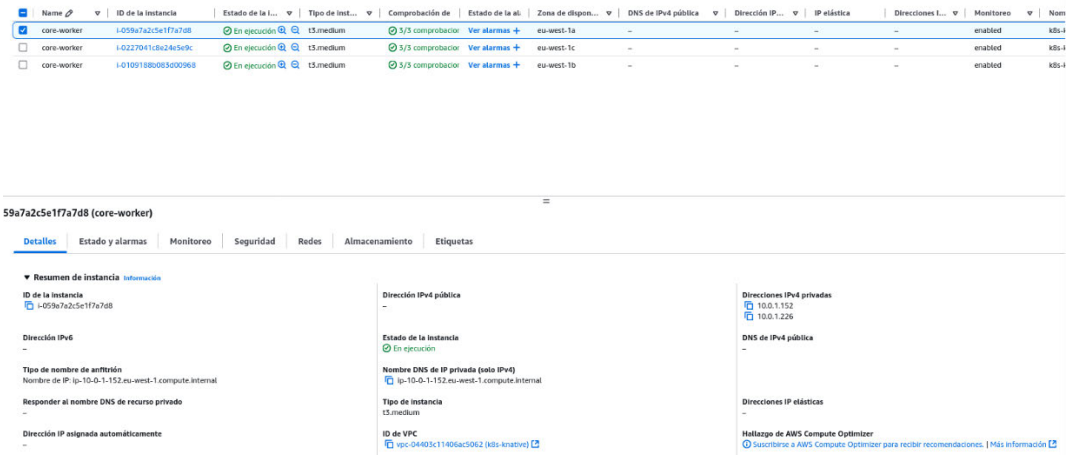
Todos los valores y configuración están parametrizados, siendo posible editar y adaptar el stack para un uso propio.

Los elementos más importantes de infraestructura son los siguientes:

- **VPC (Virtual Private Cloud):** Una red virtual en la nube que proporciona aislamiento y define el rango IP para el clúster. Se ha escogido un rango 10.0.0.0/16.
- **AZ (Availabilty Zones):** Una zona de disponibilidad (AZ) es uno o más centros de datos (CPD) con alimentación, redes y conectividad redundantes en una región de AWS. Para el caso de este proyecto se despliega todo en eu-west-1, región de Irlanda de AWS, y se hace uso de tres zonas de disponibilidad (eu-west-1a, eu-west-1b, eu-west-1c)
- **IGW (Internet Gateway):** Es un componente de la VPC que permite la comunicación entre esta e internet. Admite el tráfico IPv4 e IPv6. No genera riesgos de disponibilidad ni restricciones del ancho de banda del tráfico de red. Se despliegan en subredes públicas obligatoriamente.
- **Subredes:** Dentro de la VPC, se crean subredes (privadas y públicas) para alojar los recursos del clúster. Cada subred tiene una máscara /24, y se numeran secuencialmente (10.0.1.0/24 hasta 10.0.6.0/24). Las instancias del clúster son desplegadas en subredes privadas sin acceso directo desde Internet.
- **RT (Route Tables):** Las tablas de enrutamiento contienen conjuntos de reglas, denominadas rutas, que determinan adónde se dirige el tráfico de red desde la subred o puerta de enlace. Lleva una configuración por defecto para cada subred. El conjunto de componentes de red esquematizados se muestra en la captura 18:

Captura 18. Componentes de red *(Fuente: Elaboración propia)*

- **Instancias de EC2:** Máquinas virtuales que actúan como nodos del clúster. Estos nodos ejecutan los contenedores y servicios de Kubernetes. En la captura 19 se pueden ver tres nodos base del clúster ya desplegados con mayor nivel de detalle.



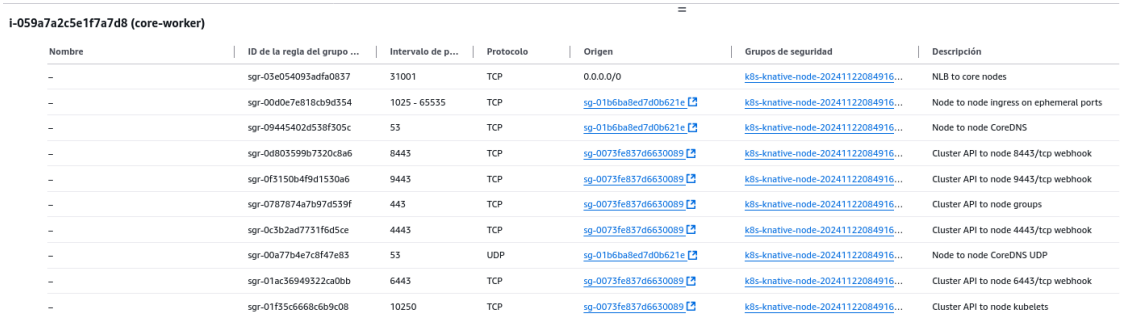
Captura 19. Detalle de nodos desplegados *(Fuente: Elaboración propia)*

Cada instancia es de tipo *t3.medium*, con las características mostradas en la figura 20. Se ha escogido este tipo de instancia buscando un balance entre el coste por hora que permitiese hacer frente al desarrollo del proyecto, y una suficiencia de recursos de hardware para hacer frente a las cargas de trabajo. Estos nodos son auto escalables horizontalmente en cada zona de disponibilidad, para ello se hace uso de un recurso de AWS llamado grupo de auto escalado, definido también con Terraform y ajustado para establecer un número mínimo (1) y máximo (3) de instancias funcionando por cada zona de disponibilidad, si bien no van a usarse más de una instancia por zona.

Nombre	vCPU	Memoria (GiB)	Rendimiento base/CPU virtual	Créditos de CPU obtenidos por hora	Ancho de banda de ráfaga de la red (Gbps)	Ancho de banda de ráfaga de EBS (Mbps)	Precio de la instancia bajo demanda por hora*
t3.nano	2	0,5	5 %	6	5	hasta 2085	0,0052 USD
t3.micro	2	1,0	10 %	12	5	hasta 2085	0,0104 USD
t3.small	2	2,0	20 %	24	5	hasta 2085	0,0209 USD
t3.medium	2	4,0	20 %	24	5	hasta 2085	0,0418 USD
t3.large	2	8,0	30 %	36	5	Hasta 2780	0,0835 USD
t3.xlarge	4	16,0	40 %	96	5	Hasta 2780	0,1670 USD
t3.2xlarge	8	32,0	40 %	192	5	Hasta 2780	0,3341 USD

Figura 20. Familia de instancias T3 *(Fuente: Amazon)*

- **SG (Security Groups):** Conjunto de reglas de cortafuegos que controlan el tráfico de entrada y salida hacia las instancias EC2. Se pueden establecer reglas de entrada y salida usando direccionamiento IP y referencias a otros grupos de seguridad de otras máquinas. Para este proyecto solo se permite tráfico de salida originario de los pods, para poder descargar imágenes de Docker, y tráfico dentro de la misma VPC y entre los pods del clúster. En la captura 21 se ejemplifica el contenido de un SG modificado durante las pruebas de despliegue.

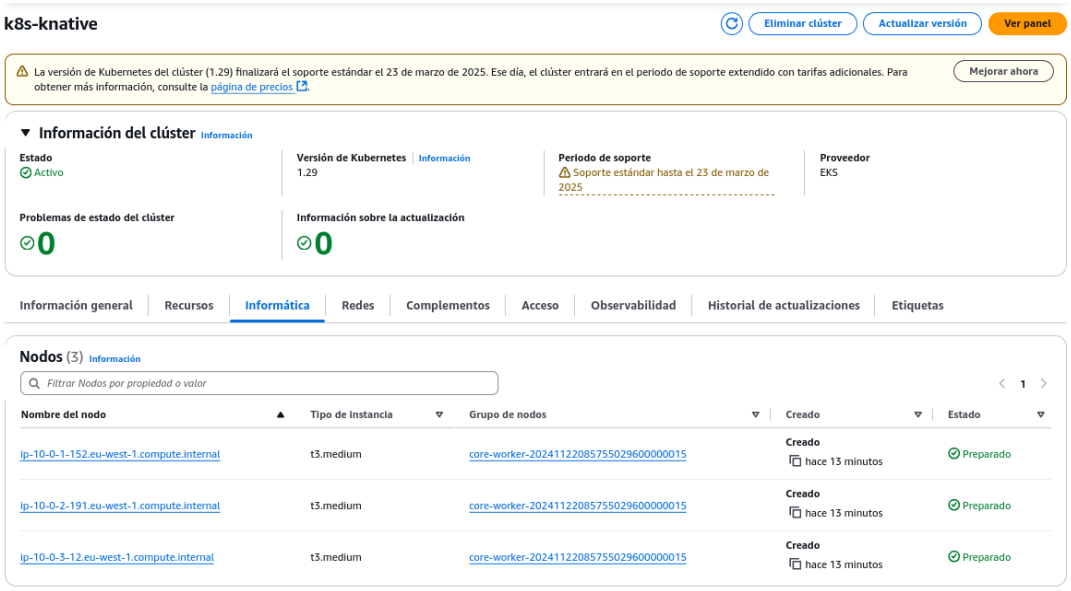


i-059a7a2c5e1f7a7d8 (core-worker)

Nombre	ID de la regla del grupo ...	Intervalo de p...	Protocolo	Origen	Grupos de seguridad	Descripción
–	sgr-03e054093adfa0837	31001	TCP	0.0.0.0/0	k8s-knative-node-20241122084916...	NLB to core nodes
–	sgr-00d0e7e818cb9d354	1025 - 65535	TCP	sg-01b6ba8ed7d0b621e	k8s-knative-node-20241122084916...	Node to node ingress on ephemeral ports
–	sgr-09445402d538f305c	53	TCP	sg-01b6ba8ed7d0b621e	k8s-knative-node-20241122084916...	Node to node CoreDNS
–	sgr-0d803599b7320c8a6	8443	TCP	sg-0073fe837d6630089	k8s-knative-node-20241122084916...	Cluster API to node 8443/tcp webhook
–	sgr-0f3150b4f9d1530a6	9443	TCP	sg-0073fe837d6630089	k8s-knative-node-20241122084916...	Cluster API to node 9443/tcp webhook
–	sgr-0787874a7b97d539f	443	TCP	sg-0073fe837d6630089	k8s-knative-node-20241122084916...	Cluster API to node groups
–	sgr-0c3b2ad7731f6d5ce	4443	TCP	sg-0073fe837d6630089	k8s-knative-node-20241122084916...	Cluster API to node 4443/tcp webhook
–	sgr-00a77b4e7c8f47e83	53	UDP	sg-01b6ba8ed7d0b621e	k8s-knative-node-20241122084916...	Node to node CoreDNS UDP
–	sgr-01ac36949322ca0bb	6443	TCP	sg-0073fe837d6630089	k8s-knative-node-20241122084916...	Cluster API to node 6443/tcp webhook
–	sgr-01f35c6668c6b9c08	10250	TCP	sg-0073fe837d6630089	k8s-knative-node-20241122084916...	Cluster API to node kubelets

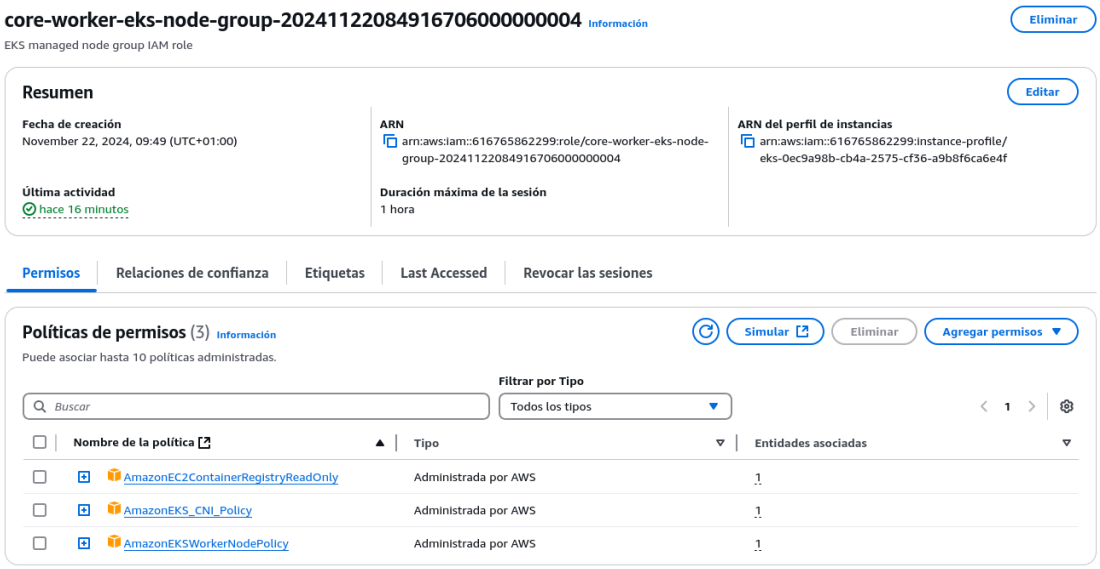
Captura 21. Reglas de seguridad (Firewall) ***(Fuente: Elaboración propia)***

- **EKS (Elastic Kubernetes Service):** En AWS, se puede utilizar el servicio gestionado de Kubernetes, EKS, que simplifica la creación y gestión de clústeres de Kubernetes. Delega el plano de control en el proveedor y abstrae de su operación a los administradores o desarrolladores. Es un recurso que dispone de un elevado número de ajustes, muy personalizable. Uno de los más relevantes es la versión del API de Kubernetes a utilizar, que para este proyecto ha sido siempre la última disponible; y en segundo lugar la disponibilidad del *endpoint* del API de Kubernetes. Para entornos críticos se recomienda el uso de un *endpoint* privado y hacer uso de una VPN (Virtual Private Network) o una solución similar. Para este proyecto se hace uso de un *endpoint* público. En la captura 22 puede observarse el interfaz web de AWS en el que se lista información del clúster ya desplegado, llamado k8s-knative.



Captura 22. Clúster de Kubernetes *(Fuente: Elaboración propia)*

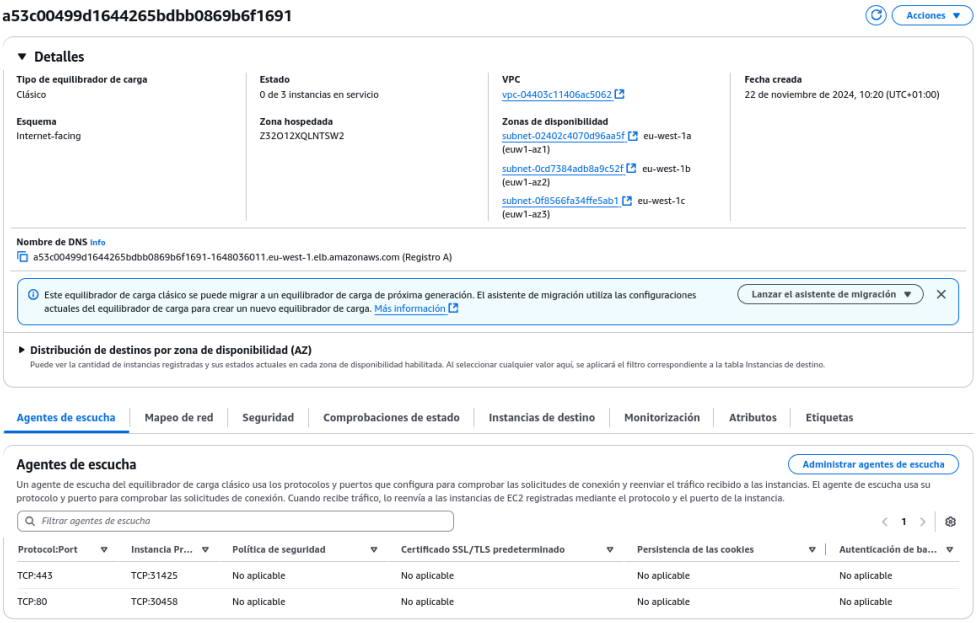
- **IAM (Identity and Access Management) Roles and Policies:** Roles y políticas (Captura 23) que otorgan los permisos necesarios para que los nodos y servicios del clúster interactúen con otros servicios de AWS vía API.



Captura 23. Ejemplo de rol con permisos en el API de AWS *(Fuente: Elaboración propia)*

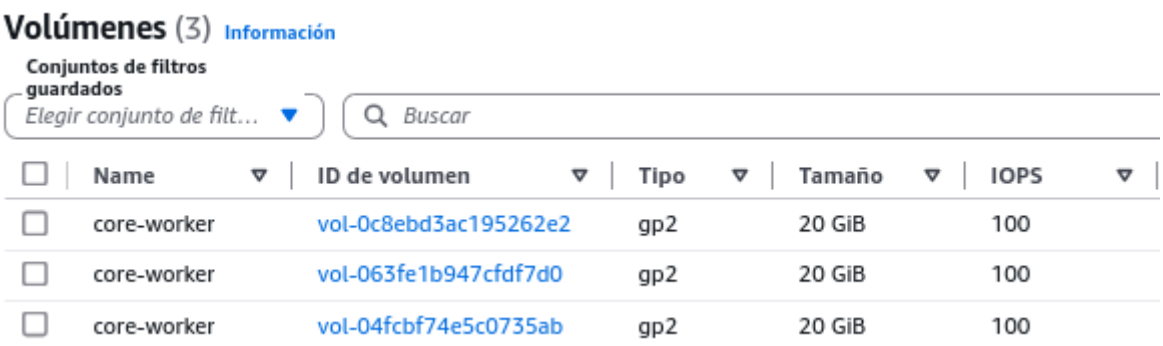
- **Repartidor de Carga:** Para distribuir el tráfico hacia los servicios desplegados en el clúster, se utilizan repartidores o balanceadores de carga como el ELB (Elastic Load Balancer). En el presente proyecto, este recurso no se crea con Terraform, sino que se delega su construcción vía API a Knative. El repartidor de carga se despliega en subred

pública y permite hacer de punto de entrada a los pods del clúster, siempre que se configure para ello. Los agentes de escucha en el puerto 443 y 80 permiten tráfico tipo HTTP directo al balanceador, que es redirigido a un *target group* o grupo de destino. Este grupo de destino está compuesto por los nodos del clúster de Kubernetes, en puertos no reservados de rango más elevado. En la captura 24 puede verse el balanceador de carga ya desplegado.



Captura 24. Panel de información de un balanceador de carga *(Fuente: Elaboración propia)*

- **Almacenamiento:** Volúmenes de almacenamiento EBS (Elastic Block Store) para el almacenamiento persistente de las instancias EC2. Son volúmenes pequeños porque la mayoría de los datos ocupan muy poco y son lo más efímeros posibles, haciendo apenas uso de ellos. Su pago es por uso en GiB, por lo que se han establecido de una capacidad muy reducida. El tipo de almacenamiento es GP2 (General Purpose 2), tipo de almacenamiento de AWS que hace uso de unidades de estado sólido con propósito de carácter general, sin carga intensiva de trabajo (Captura 25)



Captura 25. Volúmenes de almacenamiento *(Fuente: Elaboración propia)*

3.1.1 Despliegue de infraestructura con Terraform

Estos recursos se definen y gestionan de manera programática, lo que permite el aprovisionamiento de toda la infraestructura de manera automática, bajo demanda y sin cambios en la configuración no deseados.

```
101
102  module "eks_role" {
103    source = "terraform-aws-modules/iam/aws//modules/iam-role-for-service-accounts-eks"
104
105    role_name             = "eks_role"
106    attach_ebs_csi_policy = true
107    attach_vpc_cni_policy = true
108    vpc_cni_enable_ipv4   = true
109
110    oidc_providers = {
111      ex = {
112        provider_arn               = module.eks.oidc_provider_arn
113        namespace_service_accounts = ["kube-system:ebs-csi-controller-sa", "kube-system:aws-node"]
114      }
115    }
116
117    tags = local.tags
118  }
119
```

Captura 26. Módulo de Terraform *(Fuente: Elaboración propia)*

En este proyecto, como se menciona anteriormente, se ha hecho uso de módulos de Terraform ya existentes de terceros (Captura 26) para no dedicar tiempo no relacionado con el proyecto en sí y abstraer la complejidad de una arquitectura de estas características, si bien se han tenido en cuenta todas las variables y atributos configurables para desplegar las piezas deseadas. El *stack* que une estos módulos es desarrollo propio.

La ejecución de Terraform desde el PC del usuario es directa. Es necesario especificar un fichero de variables (disponible en la carpeta *environments* del stack). Al igual que con cualquier operación que implique el uso del API de AWS, es necesario tener las credenciales (Access Key y Secret Access Key) debidamente configuradas en el PC previamente.

Durante la ejecución de la herramienta, se puede ver cómo esta coteja y busca diferencias entre la entrada del usuario y el estado de la infraestructura previo. Este estado se ve reflejado en un fichero de texto tipo JSON, pero con extensión propia (*tfstate*), almacenado en la nube, en un almacenamiento de objetos (AWS S3). Este almacenamiento es creado previamente antes de desplegar toda la infraestructura.

Si Terraform encuentra diferencias entre lo solicitado y lo existente, tomará las acciones necesarias teniendo en cuenta todas las dependencias que incurren en los cambios, informando al usuario en el acto.

En el caso de una primera ejecución, esto no es necesario, pues no existen recursos creados con anterioridad. De igual modo, el fichero de estado *tfstate* será creado de cero. En la captura 27 puede verse este paso:

```
 main [x!?] using ☁ diegombeltran
in k8s-knative → terraform apply -refresh=true -var-file=environments/diego.tfvars
module.eks_role.data.aws_region.current: Reading...
module.eks.module.eks_managed_node_group["core"].data.aws_iam_policy_document.assume_role_policy[0]: Reading...
module.eks.data.aws_partition.current: Reading...
module.eks.data.aws_caller_identity.current: Reading...
module.eks_role.data.aws_partition.current: Reading...
module.eks.module.eks_managed_node_group["core"].data.aws_caller_identity.current: Reading...
module.vpc.aws_vpc.this[0]: Refreshing state... [id=vpc-04403c11406ac5062]
data.aws_caller_identity.current: Reading...
module.eks.module.eks_managed_node_group["core"].data.aws_partition.current: Reading...
module.eks_role.data.aws_partition.current: Read complete after 0s [id=aws]
module.eks_role.data.aws_region.current: Read complete after 0s [id=eu-west-1]
module.eks.data.aws_partition.current: Read complete after 0s [id=aws]
module.eks.module.eks_managed_node_group["core"].data.aws_iam_policy_document.assume_role_policy[0]: Read complete after 0s [id=2560088296]
module.eks.module.eks_managed_node_group["core"].data.aws_partition.current: Read complete after 0s [id=aws]
module.eks.data.aws_iam_policy_document.assume_role_policy[0]: Reading...
data.aws_availability_zones.available: Reading...
module.eks_role.data.aws_caller_identity.current: Reading...
module.eks.data.aws_iam_policy_document.assume_role_policy[0]: Read complete after 0s [id=2764486067]
module.eks.module.kms.data.aws_partition.current[0]: Reading...
module.eks_role.data.aws_iam_policy_document.vpc_cni[0]: Reading...
module.eks.module.kms.data.aws_partition.current[0]: Read complete after 0s [id=aws]
module.eks.module.kms.data.aws_caller_identity.current[0]: Reading...
module.eks.module.eks_managed_node_group["core"].data.aws_caller_identity.current: Read complete after 0s [id=616765862299]
module.eks_role.data.aws_iam_policy_document.vpc_cni[0]: Read complete after 0s [id=1079910255]
module.eks_role.data.aws_iam_policy_document.ebs_csi[0]: Reading...
module.eks_role.data.aws_iam_policy_document.ebs_csi[0]: Read complete after 0s [id=435063099]
module.eks.data.aws_caller_identity.current: Read complete after 0s [id=616765862299]
module.eks.data.aws_iam_session_context.current: Reading...
module.eks.data.aws_iam_session_context.current: Read complete after 0s [id=arn:aws:iam::616765862299:user/diegombeltran]
data.aws_caller_identity.current: Read complete after 0s [id=616765862299]
module.eks_role.data.aws_caller_identity.current: Read complete after 0s [id=616765862299]
module.eks.module.kms.data.aws_caller_identity.current[0]: Read complete after 0s [id=616765862299]
data.aws_availability_zones.available: Read complete after 0s [id=eu-west-1]

Terraform used the selected providers to generate the
following execution plan. Resource actions are indicated
with the following symbols:
  + create
 <= read (data resources)

Terraform will perform the following actions:

  # aws_kms_key.knative will be created
  + resource "aws_kms_key" "knative" {
      + arn                                = (known after apply)
      + bypass_policy_lockout_safety_check = false
      + customer_master_key_spec           = "SYMMETRIC_DEFAULT"
      + description                        = "k8s-knative"
      + enable_key_rotation                = false
```

Captura 27. Creación de infraestructura con Terraform *(Fuente: Elaboración propia)*

En este proyecto se crean un total de 67 recursos de infraestructura de todos los tipos indicados anteriormente, así como algunos menos relevantes de carácter auxiliar, como una clave de encriptación simétrica para la encriptación de secretos de Kubernetes dentro del clúster, y una IP estática reservada.

El despliegue de la infraestructura toma un intervalo de tiempo entre los 5 y 10 minutos, aunque es variable según la conexión. Al finalizar, es necesario configurar el acceso al API de Kubernetes para poder operar desde el PC del usuario, algo que se puede hacer con:

```
aws eks update-kubeconfig --region eu-west-1 --name k8s-knative
```

Esta orden genera un fichero en el directorio principal del usuario:
/home/usuario/.kube/config

```yaml
apiVersion: v1
clusters:
- cluster:
    certificate-authority-data: cambiadoporseguridadNDQWUyZ0F3SUJBZ0lJT0xyc1hFODJGUjB3RFFZSktv
    server: https://cambiadoporseguiridad.gr7.eu-west-1.eks.amazonaws.com
  name: arn:aws:eks:eu-west-1:cambiadoporseguridad:cluster/k8s-knative
contexts:
- context:
    cluster: arn:aws:eks:eu-west-1:cambiadoporseguridad:cluster/k8s-knative
    user: arn:aws:eks:eu-west-1:cambiadoporseguridad:cluster/k8s-knative
  name: arn:aws:eks:eu-west-1:cambiadoporseguriduad:cluster/k8s-knative
current-context: arn:aws:eks:eu-west-1:cambiadoporseguridad:cluster/k8s-knative
kind: Config
preferences: {}
users:
- name: arn:aws:eks:eu-west-1:cambiadoporseguridad:cluster/k8s-knative
  user:
    exec:
      apiVersion: client.authentication.k8s.io/v1beta1
      args:
      - --region
      - eu-west-1
      - eks
      - get-token
      - --cluster-name
      - k8s-knative
      - --output
      - json
      command: aws
      env:
      - name: AWS_PROFILE
        value: diegombeltran
```

Captura 28. Fichero .kube/config *(Fuente: Elaboración propia)*

La captura 28 es el resultado de la orden ejecutada: un fichero de texto tipo YAML sin extensión, con información sobre la dirección del API o *Control Plane* de Kubernetes (*Endpoint* de Amazon), el certificado de este, así como el método de autenticación del usuario. En este caso se especifica el uso de un perfil de AWS llamado *diegombeltran* definido en otro fichero con sus respectivos datos de autenticación para poder operar vía API con AWS, así como la orden y argumentos a utilizar por debajo. Se elimina el número de cuenta de AWS por ser un dato sensible.

Tras realizar estas acciones, es posible operar con el clúster de Kubernetes y finaliza la primera etapa de despliegue de infraestructura. En la captura 29 se hace uso del cliente *kubectl* para operar con el API del clúster. En este caso se listan los nodos.

```
 main [x!?] using ☁ diegombeltran
in k8s-knative → kubectl get nodes -o wide
NAME                                       STATUS   ROLES    AGE   VERSION               INTERNAL-IP
RSION                 CONTAINER-RUNTIME
ip-10-0-1-152.eu-west-1.compute.internal   Ready    <none>   62m   v1.29.10-eks-94953ac   10.0.1.152
219.884.amzn2.x86_64   containerd://1.7.23
ip-10-0-2-191.eu-west-1.compute.internal   Ready    <none>   61m   v1.29.10-eks-94953ac   10.0.2.191
219.884.amzn2.x86_64   containerd://1.7.23
ip-10-0-3-12.eu-west-1.compute.internal    Ready    <none>   61m   v1.29.10-eks-94953ac   10.0.3.12
219.884.amzn2.x86_64   containerd://1.7.23
```

Captura 29. Nodos del clúster de Kubernetes tras despliegue *(Fuente: Elaboración propia)*

3.2 Instalación de Knative

A partir de este punto, toda la operación se realiza dentro del clúster de Kubernetes. No es necesario modificar recursos de infraestructura de forma manual.

Existen diversas formas de instalar Knative en el clúster de Kubernetes. Para este proyecto se ha escogido la opción del operador.

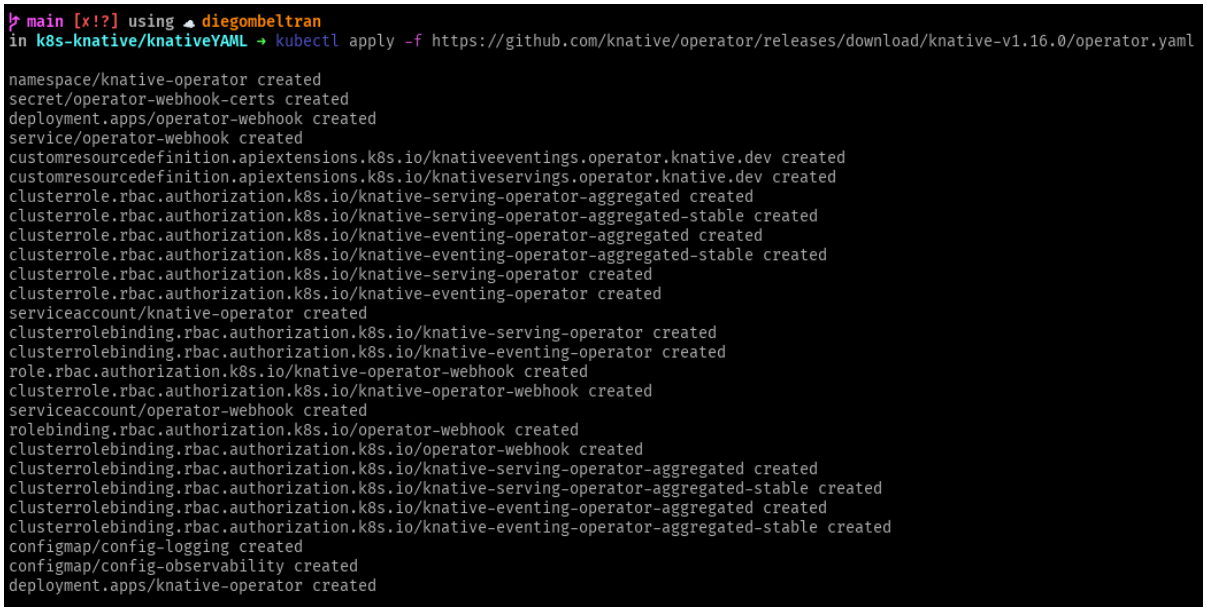
Un operador de Kubernetes es un método de empaquetar, implementar y gestionar una aplicación. Es un "asistente automático" que ayuda a manejar aplicaciones en un clúster de Kubernetes, que actúa no sólo como “receta”, sino como “cocinero”.

Este método de instalación se caracteriza por ser automatizado, por hacer uso de CRD (Custom Resource Definitions) y por tanto de expandir la capacidad de Kubernetes, y por ser capaz de gestionar dichos recursos sin mucha intervención del usuario.

La instalación del operador de Knative se puede realizar con el siguiente comando:

```
kubectl apply -f https://github.com/knative/operator/releases/download/knative-v1.16.0/operator.yaml
```

El cual crea los siguientes recursos de la captura 30:



Captura 30. Despliegue de operador de Knative *(Fuente: Elaboración propia)*

Este fichero YAML tiene su origen en el repositorio oficial de Knative en GitHub, y se ha comprobado previamente su contenido, por motivos de seguridad.

Los recursos creados son los siguientes:

- **Namespace:** Un espacio de nombres para organizar y aislar recursos dentro de un clúster de Kubernetes.
- **Secret:** Un recurso para almacenar y gestionar información sensible, como contraseñas o tokens.
- **Configmap:** Almacena datos de configuración en pares clave-valor que son utilizados por aplicaciones en el clúster.
- **Deployment:** Un controlador que gestiona la creación y escalado de conjuntos de pods, asegurando un estado deseado
- **Service:** Un recurso que define una forma de acceder a un conjunto de pods como un único servicio de red
- **CustomResourceDefinition:** Permite extender Kubernetes definiendo tipos de recursos personalizados.
- **Clusterrole.rbac.authorization:** Define un conjunto de permisos a nivel del clúster que pueden ser asignados a usuarios o recursos.
- **Serviceaccount:** Proporciona una identidad para pods que va a interactuar con el API de Kubernetes.
- **Clusterrolebinding.rbac.authorization.k8s.io:** Asocia un ClusterRole con un usuario, grupo o ServiceAccount, permitiendo permisos a nivel de clúster.

Es buena idea entender el funcionamiento de estos recursos, pero no es necesario. El operador es capaz de gestionarlos sin interacción del usuario, añadiendo una capa de abstracción extra.

El siguiente paso es instalar los dos grandes componentes de Knative, *eventing* (Captura 31) y *serving* (Captura 32). En este caso, se han descargado los YAML para personalizarlos un poco más. Por un lado, se ha añadido un namespace para cada uno de ellos con el fin de organizar los componentes de Knative en el clúster. En el caso del componente *serving,* se especifica el uso de *Kourier* (Captura 33) como capa de red por su sencillez en comparación con la alternativa propuesta por el proyecto Knative para clústers de producción, *Istio*.

Captura 31. Componente *eventing* de Knative *(Fuente: Elaboración propia)*

```yaml
apiVersion: v1
kind: Namespace
metadata:
  name: knative-serving
---
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
---
apiVersion: operator.knative.dev/v1beta1
kind: KnativeServing
metadata:
  name: knative-serving
  namespace: knative-serving
spec:
  # ...
  ingress:
    kourier:
      enabled: true
  config:
    network:
      ingress-class: "kourier.ingress.networking.knative.dev"
```

Captura 32. Componente *serving* de Knative *(Fuente: Elaboración propia)*

Al desplegar estos dos componentes, el operador de Knative previamente instalado es capaz de interpretar los objetos de tipo *apiVersion: operator.knative.dev/v1beta1*, y creará los recursos de Kubernetes nativos necesarios adicionales, sin interacción extra por parte del usuario.

La personalización del componente *serving* para hacer uso de Kourier hace una llamada al API de Kubernetes para crear un servicio de tipo LoadBalancer, que a su vez hace otra llamada al API AWS para crear otro recurso de infraestructura de tipo Load Balancer. El registro DNS de dicho balanceador se asocia al recurso de Kubernetes.

De ahora en adelante, es posible realizar peticiones al clúster vía puerto 80 o puerto 443 desde Internet a través del balanceador de carga, y las peticiones serán distribuidas entre los nodos del clúster de Kubernetes. En la práctica esto no es posible porque no se desea tráfico externo con origen desde Internet al clúster por motivos de seguridad, por lo que se han cerrado las reglas pertinentes en los correspondientes grupos de seguridad (Security Groups).

```
 main [x!?] using ☁ diegombeltran
in k8s-knative/knativeYAML → kubectl --namespace knative-serving get service kourier
NAME       TYPE           CLUSTER-IP       EXTERNAL-IP                                                                PORT(S)                      AGE
kourier    LoadBalancer   172.20.220.244   a55308d2bedb64bcaa0da1c6a3265d05-2081965030.eu-west-1.elb.amazonaws.com   80:30407/TCP,443:31461/TCP   5m43s
```

Captura 33. Componente de red Kourier *(Fuente: Elaboración propia)*

Si se listan los pods que corren en el clúster tras la instalación de los componentes de Knative se tiene lo mostrado en la captura 34:

```
 main [x!?] using ☁ diegombeltran
in k8s-knative/knativeYAML → k get pods -A
NAMESPACE          NAME                                      READY   STATUS    RESTARTS   AGE
knative-eventing   eventing-controller-5fc484f6fd-wr4lt      1/1     Running   0          36m
knative-eventing   eventing-webhook-6485bcc49d-fh45p         1/1     Running   0          36m
knative-eventing   imc-controller-d7c5d5647-h5fc5            1/1     Running   0          36m
knative-eventing   imc-dispatcher-75fb864995-9qvlz           1/1     Running   0          36m
knative-eventing   job-sink-76965754c4-9mhg9                 1/1     Running   0          36m
knative-eventing   mt-broker-controller-5d87695c8b-ll2cw     1/1     Running   0          36m
knative-eventing   mt-broker-filter-5fc49bdff7-xvb9c         1/1     Running   0          36m
knative-eventing   mt-broker-ingress-599f4466fd-m78xd        1/1     Running   0          36m
knative-operator   knative-operator-67c5f99c66-68dxr         1/1     Running   0          81m
knative-operator   operator-webhook-64c7757748-zs754         1/1     Running   0          81m
knative-serving    3scale-kourier-gateway-69dfb8cffc-6rsc9   1/1     Running   0          12m
knative-serving    activator-86df87955f-g5vn7                1/1     Running   0          13m
knative-serving    autoscaler-7f84f7c888-gxgs8               1/1     Running   0          13m
knative-serving    autoscaler-hpa-59c5b7db46-bqwrg           1/1     Running   0          13m
knative-serving    controller-8566b8848b-7p59k               1/1     Running   0          13m
knative-serving    net-kourier-controller-cc48d56db-k28fb    1/1     Running   0          12m
knative-serving    webhook-7f6b6fbd87-8fvqw                  1/1     Running   0          13m
kube-system        aws-node-b55qj                            2/2     Running   0          100m
kube-system        aws-node-l79bk                            2/2     Running   0          100m
kube-system        aws-node-z7cjs                            2/2     Running   0          100m
kube-system        coredns-7f9877d69c-gdv5t                  1/1     Running   0          100m
kube-system        coredns-7f9877d69c-xw9nl                  1/1     Running   0          100m
kube-system        ebs-csi-controller-f5694ccd-4dmds         6/6     Running   0          100m
kube-system        ebs-csi-controller-f5694ccd-d7zhm         6/6     Running   0          100m
kube-system        ebs-csi-node-55lnf                        3/3     Running   0          100m
kube-system        ebs-csi-node-jlts6                        3/3     Running   0          100m
kube-system        ebs-csi-node-v847w                        3/3     Running   0          100m
kube-system        kube-proxy-d9nq6                          1/1     Running   0          100m
kube-system        kube-proxy-qmrnw                          1/1     Running   0          100m
kube-system        kube-proxy-rjjcf                          1/1     Running   0          100m
```

Captura 34. Listado inicial de pods del clúster *(Fuente: Elaboración propia)*

Por un lado, se encuentran todos los pods necesarios para el correcto funcionamiento del clúster de Kubernetes, bajo el *namespace* kube-system. Estos pods y sus recursos asociados, como servicios, serviceaccounts, etc, son creados durante el despliegue de la infraestructura con Terraform.

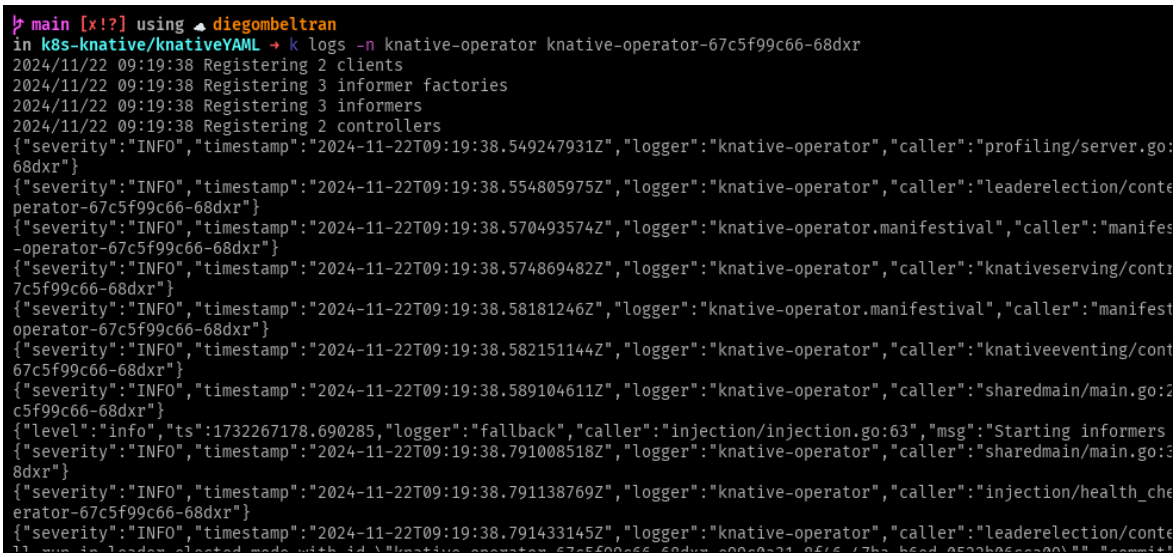
- **aws-node:** Gestiona la integración de nodos de AWS con Kubernetes, manejando la red y las direcciones IPs en clústeres de Amazon EKS.
- **coredns:** Proporciona servicios de DNS dentro del clúster, resolviendo nombres de los servicios y pods en Kubernetes.
- **ebs-csi-controller:** Gestiona la provisión y el ciclo de vida de volúmenes de Amazon EBS para el almacenamiento en clústeres de Kubernetes.
- **ebs-csi-node:** Ejecuta el componente del nodo para conectar y montar volúmenes EBS en los nodos donde se ejecutan los pods.
- **kube-proxy:** Mantiene las reglas de red en los nodos y gestiona el balanceo de carga y reenvío de tráfico para servicios en Kubernetes.

Respecto a Knative, los pods requeridos se encuentran repartidos en tres namespaces, *knative-operator*, *knative-serving* y *knative-eventing*.

No es un requisito entender completamente el funcionamiento de cada uno de los pods, pero sí es obligatorio asegurar el correcto funcionamiento de estos. Para ello es necesario comprobar que los contenedores dentro de cada pod están corriendo y en estado *ready.*

En el hipotético caso de que alguno de estos pods estuviese indisponible, sería necesario diagnosticar el motivo. Una forma puede ser revisando los logs (Captura 35) de cada uno de ellos:

```
kubectl logs -n knative-operator knative-operator-67c5f99c66-68dxr
```



Captura 35. Ejemplo de logs del operador de Knative *(Fuente: Elaboración propia)*

3.3 Adaptación de código fuente a Julia

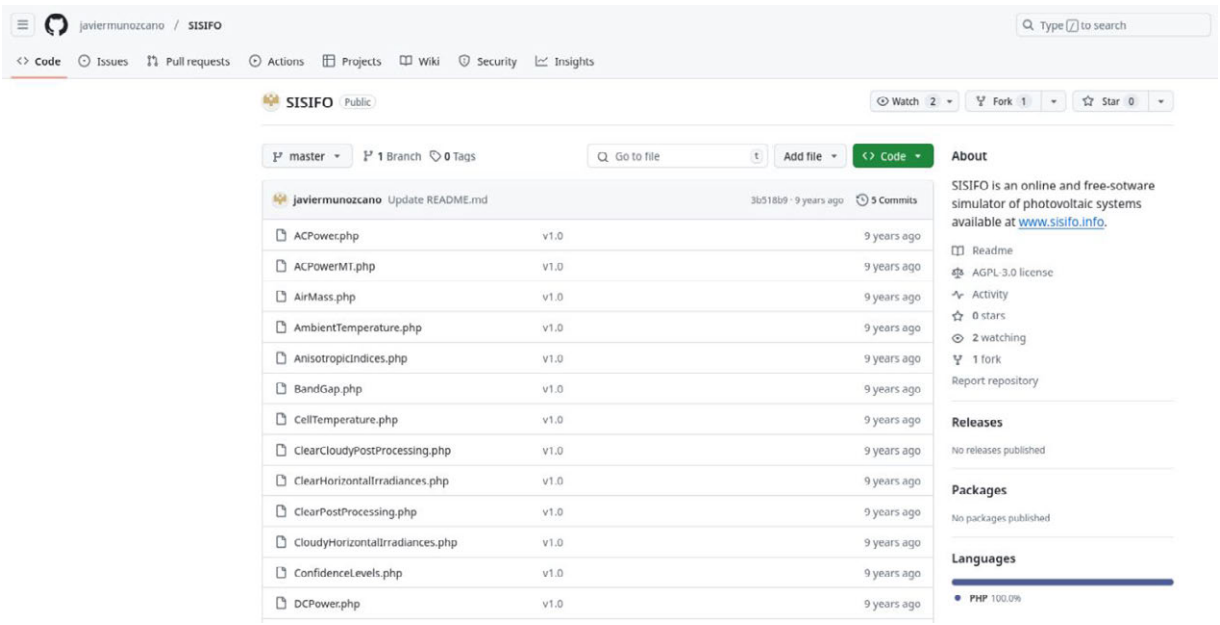
En el desarrollo del proyecto, uno de los principales puntos a abordar ha sido la migración del código fuente del simulador SISIFO, originalmente escrito en PHP, al lenguaje de programación Julia. Este proceso incluye la transcripción del código, el uso de características propias del nuevo lenguaje y un intento de optimización en las funciones utilizadas para tratar con los datos.

El proceso de adaptación se ha realizado en diferentes etapas descritas a continuación.

3.3.1 Análisis de código existente

Como se cuenta en la sección preliminar de introducción, el proyecto usa una versión de SISIFO traducida desde MATLAB a PHP en 2015, algo más simple que la actual, pero completamente funcional. La interfaz web actual está escrita en C# y accede a un microservicio que permite el uso concurrente de varios motores MATLAB para la ejecución del código. La estructura de funciones es la misma, y sirve para comparar, solo que la traducción a PHP implica la pérdida de la capacidad de cálculo matricial.

En primera instancia se hace necesario un análisis de la estructura del código, disponible públicamente con licencia AGPLv3 en GitHub (Captura 36):

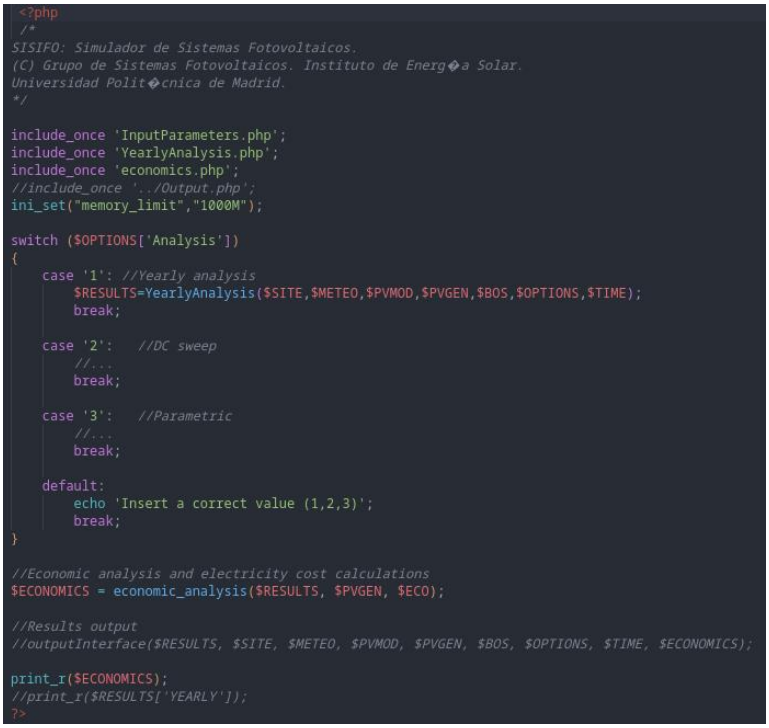


Captura 36. Repositorio público de SISIFO *(Fuente: Elaboración propia)*

La estructura se compone de un conjunto de ficheros PHP que contienen las diferentes funciones que van a ser invocadas, con una relación de dependencias basada en instrucciones *include*.

De estos ficheros existen dos a destacar:

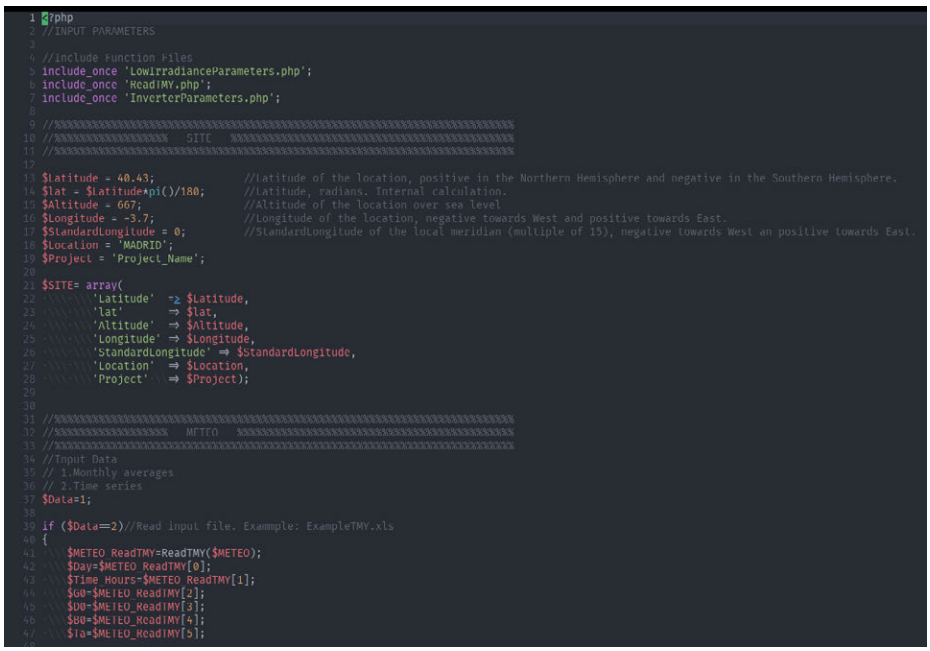
- **SISIFO.php**: Punto de entrada o *método main.* Es el bloque para realizar una secuencia completa y obtener los datos deseados a partir de unos parámetros de entrada (Captura 37).



```php
<?php
/*
SISIFO: Simulador de Sistemas Fotovoltaicos.
(C) Grupo de Sistemas Fotovoltaicos. Instituto de Energ�a Solar.
Universidad Polit�cnica de Madrid.
*/

include_once 'InputParameters.php';
include_once 'YearlyAnalysis.php';
include_once 'economics.php';
//include_once '../Output.php';
ini_set("memory_limit","1000M");

switch ($OPTIONS['Analysis'])
{
    case '1': //Yearly analysis
        $RESULTS=YearlyAnalysis($SITE,$METEO,$PVMOD,$PVGEN,$BOS,$OPTIONS,$TIME);
        break;

    case '2':   //DC sweep
        //...
        break;

    case '3':   //Parametric
        //...
        break;

    default:
        echo 'Insert a correct value (1,2,3)';
        break;
}

//Economic analysis and electricity cost calculations
$ECONOMICS = economic_analysis($RESULTS, $PVGEN, $ECO);

//Results output
//outputInterface($RESULTS, $SITE, $METEO, $PVMOD, $PVGEN, $BOS, $OPTIONS, $TIME, $ECONOMICS);

print_r($ECONOMICS);
//print_r($RESULTS['YEARLY']);
?>
```

Captura 37. SISIFO.php *(Fuente: Elaboración propia)*

- **InputParameters.php**: Definición de los parámetros de entrada, así como asignación de valores de muestra. Estos parámetros son los que un usuario del simulador introduciría para obtener la simulación. Son los datos que se han manejado en el desarrollo del proyecto. En la captura 38 puede verse el formato:

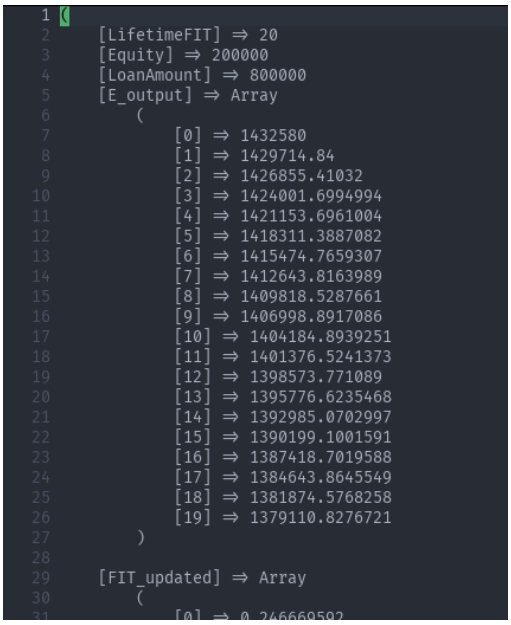


```php
<?php
//INPUT PARAMETERS

//include Function Files
include_once 'LowirradianceParameters.php';
include_once 'ReadTMY.php';
include_once 'InverterParameters.php';

//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%   SITE   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%

$latitude = 40.43;              //Latitude of the location, positive in the Northern Hemisphere and negative in the Southern Hemisphere.
$lat = $latitude*pi()/180;      //Latitude, radians. Internal calculation.
$Altitude = 667;                //Altitude of the location over sea level
$Longitude = -3.7;              //Longitude of the location, negative towards West and positive towards East.
$StandardLongitude = 0;         //StandardLongitude of the local meridian (multiple of 15), negative towards West an positive towards East.
$Location = 'MADRID';
$Project = 'Project_Name';

$SITE= array(
        'Latitude'  => $Latitude,
        'lat'       => $lat,
        'Altitude'  => $Altitude,
        'Longitude' => $Longitude,
        'StandardLongitude' => $StandardLongitude,
        'Location'  => $Location,
        'Project'   => $Project);


//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%   METEO   %%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%%
//Tnput Data
// 1.Monthly averages
// 2.Time series
$Data=1;

if ($Data==2)//Read input file. Exammple: ExampleTMY.xls
{
    $METEO_ReadTMY=ReadTMY($METEO);
    $Day=$METEO_ReadTMY[0];
    $Time_Hours=$METEO_ReadTMY[1];
    $G0=$METEO_ReadTMY[2];
    $D0=$METEO_ReadTMY[3];
    $B0=$METEO_ReadTMY[4];
    $Ta=$METEO_ReadTMY[5];
```

Captura 38. InputParameters.php *(Fuente: Elaboración propia)*

3.3.2 Uso de datos de muestra

Una ejecución de esta versión de SISIFO toma en cuenta los parámetros de entrada y devuelve un fichero de 446 líneas con el resultado de la simulación. En la captura 39 puede verse un fragmento de la salida impresa con la función *print_r* de PHP. En este caso el contenido de E_Output (Predicción de energía a la salida del generador FV durante los próximos 20 años)



```
Array
(
    [LifetimeFIT] => 20
    [Equity] => 200000
    [LoanAmount] => 800000
    [E_output] => Array
        (
            [0] => 1432580
            [1] => 1429714.84
            [2] => 1426855.41032
            [3] => 1424001.6994994
            [4] => 1421153.6961004
            [5] => 1418311.3887082
            [6] => 1415474.7659307
            [7] => 1412643.8163989
            [8] => 1409818.5287661
            [9] => 1406998.8917086
            [10] => 1404184.8939251
            [11] => 1401376.5241373
            [12] => 1398573.771089
            [13] => 1395776.6235468
            [14] => 1392985.0702997
            [15] => 1390199.1001591
            [16] => 1387418.7019588
            [17] => 1384643.8645549
            [18] => 1381874.5768258
            [19] => 1379110.8276721
        )

    [FIT_updated] => Array
        (
            [0] => 0.246669592
```

Captura 39. Output tras ejecución *(Fuente: Elaboración propia)*

El fichero **InputParameters.php** consiste en 667 líneas, las cuales incluyen definición y asignación de variables en muchos casos de forma dinámica y en función de selectores por parte del usuario. Por ejemplo, dependiendo del valor de variables como *$Sky* o *$Data,* el simulador tendrá en cuenta unos parámetros u otros.

Sucede lo mismo con algunas funciones: dependiendo del tipo de estructura (Variable *$struct)* la simulación invoca a una u otra para generar la matriz *PVGEN.*

Ya que en los datos de prueba no se usan muchos parámetros y otros tienen definido el camino y las funciones a invocar, como acción en esta fase se refactoriza y limpia el código antes de convertirlo, aumentando la legibilidad y facilitando su uso a posteriori.

El nuevo fichero **InputParameters.php** pasa a tener 147 líneas, consistiendo en 12 arrays.

3.3.3 Análisis de flujo y secuencia

La siguiente acción llevada a cabo es la comprensión de la secuencia llevada a cabo por el simulador desde su ejecución mediante **SISIFO.php**.

Mediante el uso de puntos de ruptura en el entorno de desarrollo y uso de *debugger*, así como de forma manual mediante el dibujo de diagramas de clases y secuencia, es posible esbozar una idea del orden de invocación de las diferentes funciones contenidas en los numerosos ficheros. Las figuras 40 y 41 muestran un diagrama de secuencia de SISIFO.php y YearlyAnalysis.php, respectivamente:

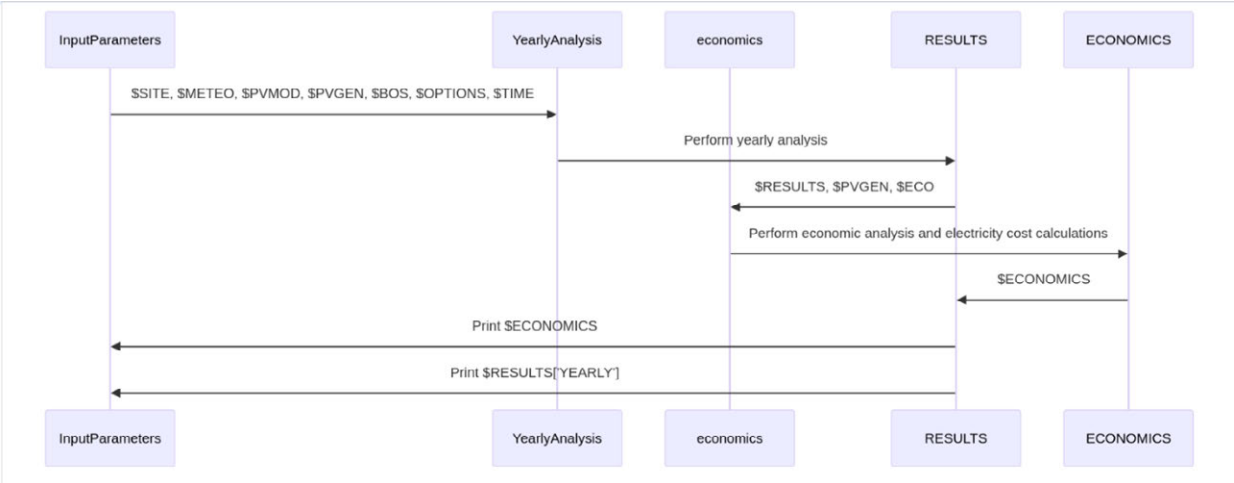


Figura 40. Diagrama de secuencia de SISIFO.php *(Fuente: Elaboración propia)*

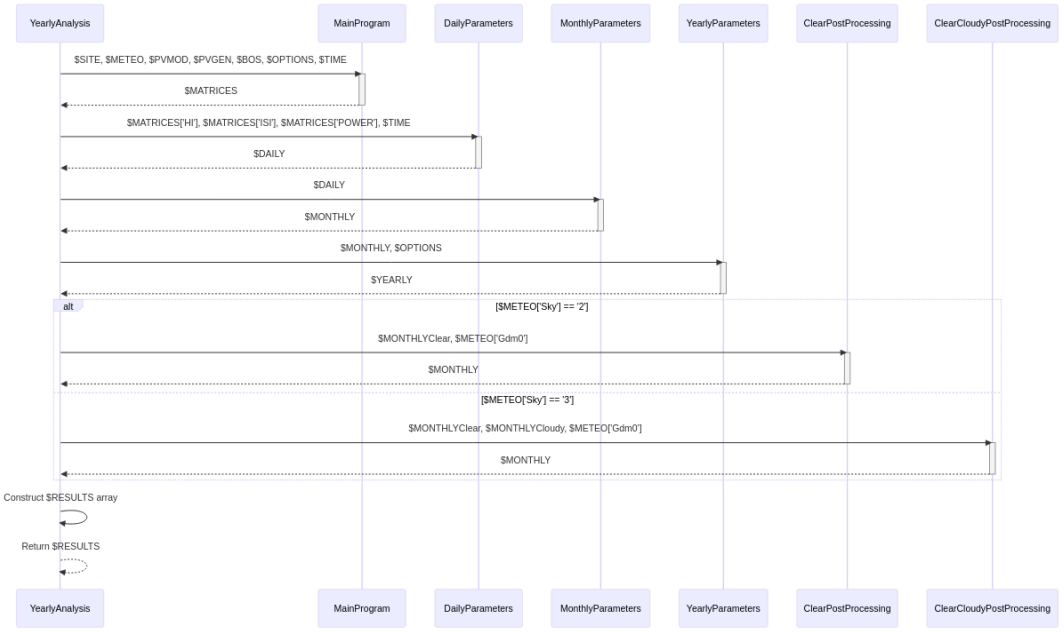


Figura 41. Diagrama de secuencia de YearlyAnalysis.php *(Fuente: Elaboración propia)*

3.3.4 Elección de fragmento del simulador a analizar

La conversión completa de esta versión del simulador a Julia y microservicios se encuentra fuera del objetivo del presente proyecto, por lo que para una demostración y prueba de funcionamiento se ha escogido uno de los caminos que sigue el código en función de los parámetros de entrada proporcionados, en particular, uno de los tipos de generador fotovoltaico: una estructura estática sobre suelo o tejado.

Teniendo en cuenta los datos de prueba proporcionados, la llamada a la función *MainProgram* (Definida en **MainProgram.php**) resulta interesante desde un punto de vista de detalle y datos aportados.

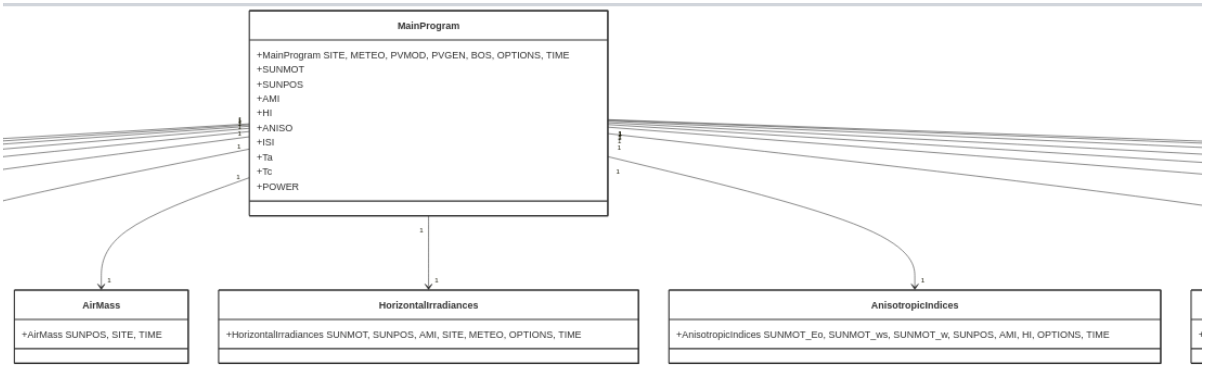


Figura 42. Diagrama de clases de MainProgram.php *(Fuente: Elaboración propia)*

Este componente (Figura 42) hace uso de la mayoría de las funciones y dependencias del simulador, así como la mayoría de los parámetros de entrada, y da como resultado una matriz con un número relevante de datos.

Antes de la conversión a Julia, se refactorizó para aumentar la legibilidad y simplificar la base de código a traducir, eliminando funciones, variables, estructuras de control y comentarios no utilizados.

En las funciones de código previas a *MainProgram* en la secuencia, se han comentado llamadas a las otras funciones no utilizadas y fuera del camino escogido.

3.3.5 Conversión a Julia

Se ha realizado la conversión de la mayoría de los ficheros PHP del repositorio a Julia, a excepción de 10 ficheros. Estos 10 ficheros son bloques de código mucho más extensos, que, por el camino escogido para el desarrollo del proyecto, no son utilizados.

La traducción a Julia es relativamente accesible. Julia es un lenguaje interpretado con una sintaxis limpia y minimalista, muy similar a MATLAB o Python (Captura 43).

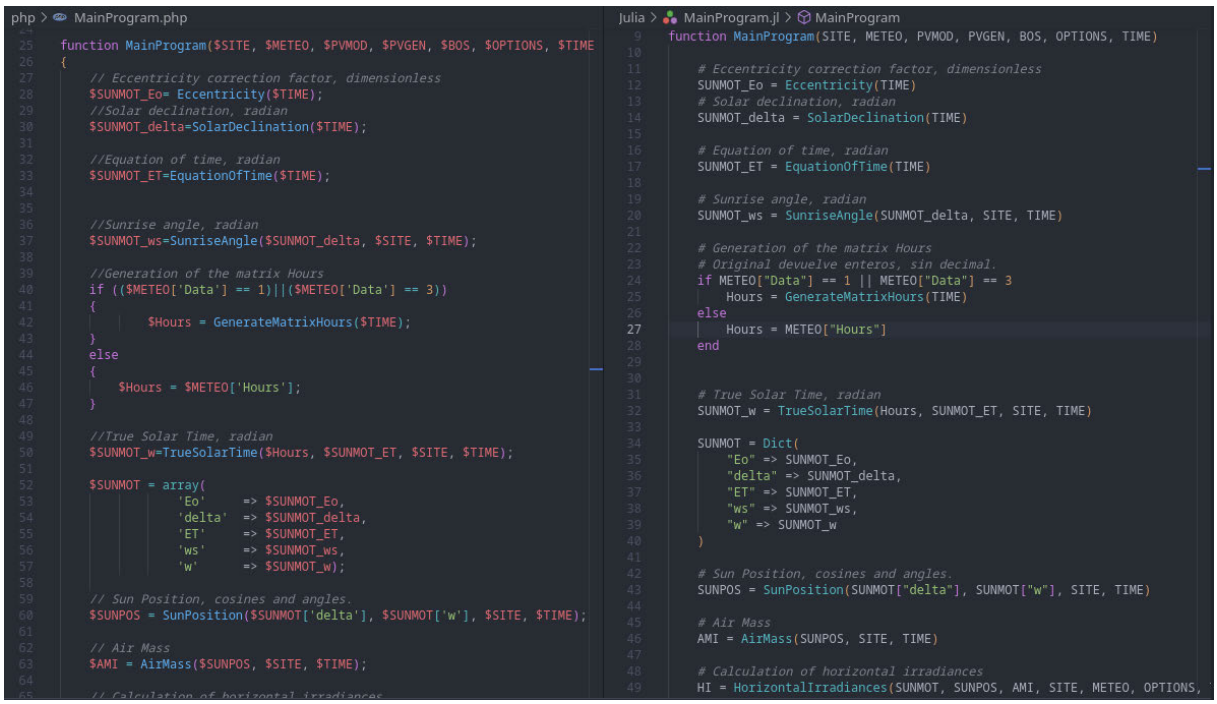
El lenguaje está orientado a las matemáticas y a la computación numérica y científica, por lo que las expresiones más complejas son fáciles de escribir.

Es tipado dinámico, por lo que no es necesario fijar el tipo de datos de las variables, no obstante, esto se explicará con más detalle en futuros apartados.

A modo de ejemplo, el fichero MainProgram.php ya refactorizado previamente a la conversión, tiene un total de 103 líneas, siendo el fichero convertido **MainProgram.jl** de 83 líneas.

En la fase de traducción y documentación usando Julia, se encontraron particularidades del lenguaje diferentes a otros con los que se tenía experiencia previa, aunque no todas estas características se usaron para el desarrollo del proyecto.

- No existen estructuras de control tipo switch/case
- Tiene un sistema de tipado dinámico y opcional
- No tiene clases. Utiliza tipos abstractos y concretos y se basa en la composición para modelar datos, promoviendo un diseño centrado en funciones.
- Se compila justo a tiempo (JIT) a código máquina a través de LLVM.
- Soporta la concurrencia, pero utiliza tareas (corutinas) para el paralelismo cooperativo, lo que es diferente del modelo de hilos tradicionales.



```php
function MainProgram($SITE, $METEO, $PVMOD, $PVGEN, $BOS, $OPTIONS, $TIME
{
    // Eccentricity correction factor, dimensionless
    $SUNMOT_Eo= Eccentricity($TIME);
    //Solar declination, radian
    $SUNMOT_delta=SolarDeclination($TIME);

    //Equation of time, radian
    $SUNMOT_ET=EquationOfTime($TIME);


    //Sunrise angle, radian
    $SUNMOT_ws=SunriseAngle($SUNMOT_delta, $SITE, $TIME);

    //Generation of the matrix Hours
    if (($METEO['Data'] == 1)||($METEO['Data'] == 3))
    {
        $Hours = GenerateMatrixHours($TIME);
    }
    else
    {
        $Hours = $METEO['Hours'];
    }

    //True Solar Time, radian
    $SUNMOT_w=TrueSolarTime($Hours, $SUNMOT_ET, $SITE, $TIME);

    $SUNMOT = array(
                'Eo'    => $SUNMOT_Eo,
                'delta' => $SUNMOT_delta,
                'ET'    => $SUNMOT_ET,
                'ws'    => $SUNMOT_ws,
                'w'     => $SUNMOT_w);

    // Sun Position, cosines and angles.
    $SUNPOS = SunPosition($SUNMOT['delta'], $SUNMOT['w'], $SITE, $TIME);

    // Air Mass
    $AMI = AirMass($SUNPOS, $SITE, $TIME);
```

```julia
function MainProgram(SITE, METEO, PVMOD, PVGEN, BOS, OPTIONS, TIME)

    # Eccentricity correction factor, dimensionless
    SUNMOT_Eo = Eccentricity(TIME)
    # Solar declination, radian
    SUNMOT_delta = SolarDeclination(TIME)

    # Equation of time, radian
    SUNMOT_ET = EquationOfTime(TIME)

    # Sunrise angle, radian
    SUNMOT_ws = SunriseAngle(SUNMOT_delta, SITE, TIME)

    # Generation of the matrix Hours
    # Original devuelve enteros, sin decimal.
    if METEO["Data"] == 1 || METEO["Data"] == 3
        Hours = GenerateMatrixHours(TIME)
    else
        Hours = METEO["Hours"]
    end


    # True Solar Time, radian
    SUNMOT_w = TrueSolarTime(Hours, SUNMOT_ET, SITE, TIME)

    SUNMOT = Dict(
        "Eo" => SUNMOT_Eo,
        "delta" => SUNMOT_delta,
        "ET" => SUNMOT_ET,
        "ws" => SUNMOT_ws,
        "w" => SUNMOT_w
    )

    # Sun Position, cosines and angles.
    SUNPOS = SunPosition(SUNMOT["delta"], SUNMOT["w"], SITE, TIME)

    # Air Mass
    AMI = AirMass(SUNPOS, SITE, TIME)

    # Calculation of horizontal irradiances
    HI = HorizontalIrradiances(SUNMOT, SUNPOS, AMI, SITE, METEO, OPTIONS,
```

Captura 43. Comparativa sintaxis PHP/Julia *(Fuente: Elaboración propia)*

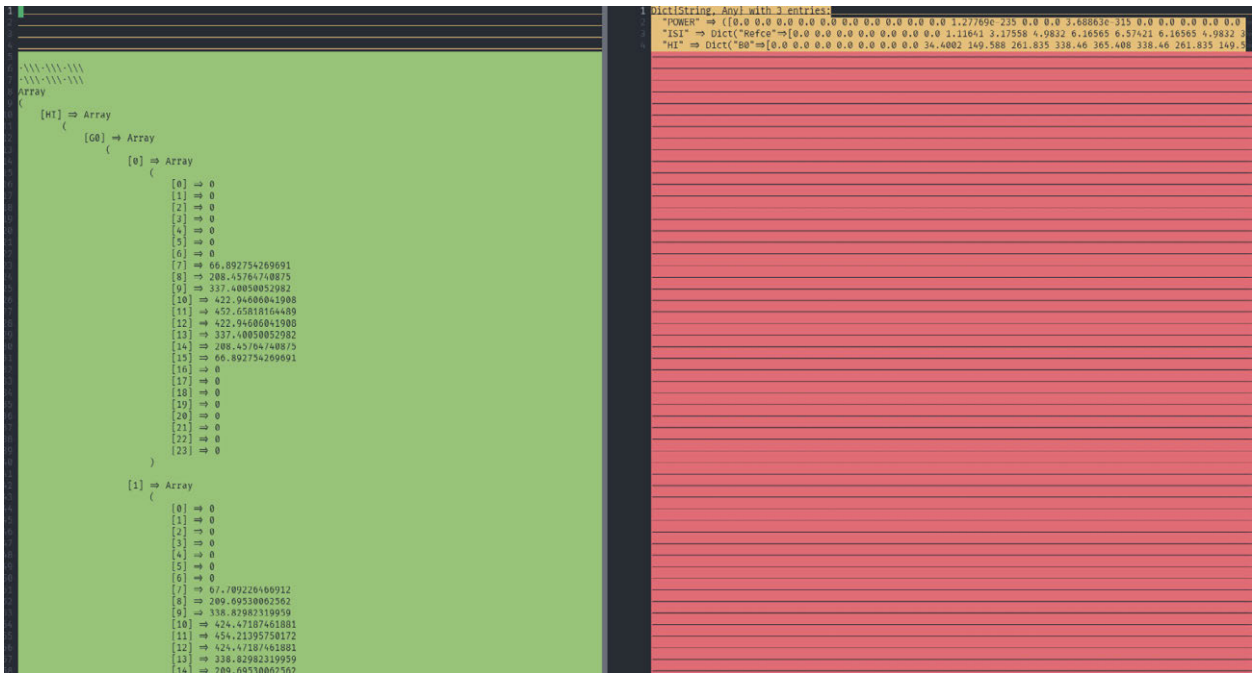
Para facilitar la organización del código y llamada a las funciones, se ha creado un paquete de Julia (Package) que hace la función de librería. De esa forma MainProgram.jl llama a SISIFOLibrary.jl, que contiene todas las funciones necesarias y sus dependencias.

En el próximo apartado se atomizarán estas funciones para convertirlas en microservicios.

3.3.6 Comprobación de funcionamiento

Tras la conversión de toda la base del código a Julia, se ejecutan ambos para comprobar que a pesar de la sintaxis es correcta y no se encuentran fallos en el entorno de desarrollo, los resultados son idénticos. Se han añadido funciones de impresión por pantalla específicas para arrays para hacer legible la salida de los numerosos datos:

```
php MainProgram.php > ../MainProgramPHPOutput.txt
julia MainProgram.jl > ../MainProgramJuliaOutput.txt
vimdiff MainProgramPHPOutput.txt MainProgramJuliaOutput.txt
```



Captura 44. Comparación output PHP/Julia *(Fuente: Elaboración propia)*

Como se observa en la captura 44, aunque a priori pueda parecer que son resultados completamente distintos, el motivo es el formato utilizado por cada una de las funciones de impresión por pantalla, print_r en el caso de PHP y display en el caso de Julia. Una observación más concreta y una búsqueda en los datos revela que son los mismos resultados.

Una cosa para tener en cuenta es que el tipado de los datos es diferente. En el caso de PHP muchos parámetros se consideran tipo entero (int), como el 0, mientras que su contraparte en Julia es, al igual que en la versión de SISIFO escrita en MATLAB, un número en coma flotante de 64 bits (float64).

Esto también se ha tenido en cuenta en las operaciones matemáticas, que en Julia tienen una mayor precisión y mayor número de decimales. No obstante, este comportamiento es configurable mediante el tipado de las variables en su definición. Julia soporta además muchos tipos primitivos.

3.3.7 Tiempo de ejecución

A modo de recopilación de datos para conclusiones, se ha repetido el experimento de ejecutar ambos lenguajes haciendo uso de la función *time*, obteniendo los siguientes resultados:

```
time php MainProgram.php
time julia MainProgram.jl
```

Tabla 1. Comparativa de primera ejecución local de MainProgram sin Oxygen

	JULIA (1 ejecución)	JULIA (2 y sucesivas)	PHP
Tiempo de usuario	8.35s	0.9s	0.41s
Tiempo de sistema	0.21s	0.07s	0.07s
Tiempo total	8.547s	0.97s	0.479s

- **Tiempo de usuario:** Tiempo de CPU consumido por el script. Es el tiempo dedicado a ejecutar las instrucciones del programa.
- **Tiempo de sistema:** Tiempo dedicado a operaciones del sistema, como operaciones de entrada/salida.
- **Tiempo total:** Total transcurrido desde que el comando comienza a ejecutarse hasta que termina, incluyendo tanto el tiempo de usuario como el del sistema y cualquier tiempo de espera incluido.

Estos tiempos muestran una diferencia significativa entre la primera ejecución de un script en Julia y el mismo script en PHP.

La ejecución con Julia es más lenta la primera vez, de forma similar a, por ejemplo, .NET [55]. Esto es debido al tiempo de compilación en tiempo de ejecución y al arranque del entorno de Julia. Julia compila el código antes de ejecutarlo, lo que puede introducir una sobrecarga considerable en scripts pequeños.

3.4 Knative para orquestación de microservicios

En este apartado, se pone el foco en la creación de microservicios comunicados vía HTTP haciendo uso de Knative.

3.4.1 Creación de microservicio web

Como primer paso para mover a una arquitectura basada en microservicios, se decide hacer una prueba de lectura de los parámetros de entrada de prueba vía HTTP.

Para cumplir dicho objetivo se plantean dos hitos:

1. Los parámetros de entrada deben ser editables de forma sencilla y ser agnósticos al código en Julia. De esa manera se pueden enviar vía petición HTTP y la función de lectura puede implementarse en cualquier lenguaje.
2. Teniendo en cuenta que en esta primera aproximación la función de entrada que lee los parámetros está desarrollada en Julia, se hace necesario buscar entre los paquetes del lenguaje para poder implementar un servidor web que acepte peticiones y permita implementar un *router* y varios endpoints para cada verbo HTTP.

En el caso de esta prueba de concepto se ha encontrado un micro-framework de Julia llamado Oxygen.jl, elaborado haciendo uso de la librería HTTP.jl, muy utilizada en Julia.

Entre las características de Oxygen.jl se encuentran funciones como la serialización y deserialización JSON listas para usar y personalizables, los extractores de solicitudes, soporte para múltiples instancias, multithreading, websocket, documentación swagger autogenerada, regeneración en caliente al modificar el código e incluso un panel de métricas en tiempo real. Ofrece mucha facilidad a la hora de implementar un pequeño microservicio web que escuche en un puerto personalizable.

El primer paso consiste en “deserializar” el fichero InputParameters.jl ya convertido a Julia, y que contiene los parámetros de entrada. El formato escogido para contener los nuevos parámetros es JSON debido a su sencillez, claridad y compatibilidad con infinidad de lenguajes y herramientas. Se muestra en la captura 45 un ejemplo de datos.

```
{
    "PVGen": {
        "Struct": 1,
        "PowNom_Total": 1000,
        "PowNom_PerInverter": 100,
        "PowNom_PerTransformer": 1000,
        "PRVPN": 1,
        "PowReal_Total": 1000,
        "NBGH": 10,
        "NBGV": 10,
        "NBT": 100,
        "GroundRoof": {
            "Inclination_roof": 0,
            "Orientation_roof": 0,
            "Inclination_generator": 35,
            "Orientation_generator": 0,
            "LNS": 1.5,
            "AEO": 5,
            "DEO": 0
        }
    },
    "Site": {
        "Latitude": 40.43,
        "lat": 0.7056366165813074,
        "Altitude": 667,
        "Longitude": -3.7,
        "StandardLongitude": 0,
        "Location": "MADRID",
        "Project": "Project_Name"
    },
    "Meteo": {
        "Data": 1,
        "Sky": 2,
        "Gdm0": [
```

Captura 45. Parámetros en formato JSON *(Fuente: Elaboración propia)*

El segundo paso para esta prueba de concepto consiste en desarrollar una pequeña función, de nombre readInputParameters.jl, que haga uso de Oxygen.jl y que sea capaz de leer el JSON anterior y volver a serializarlo.

Sin contar con la definición de estructuras (Structs) de Julia, el código queda tan contenido como en la captura 46:

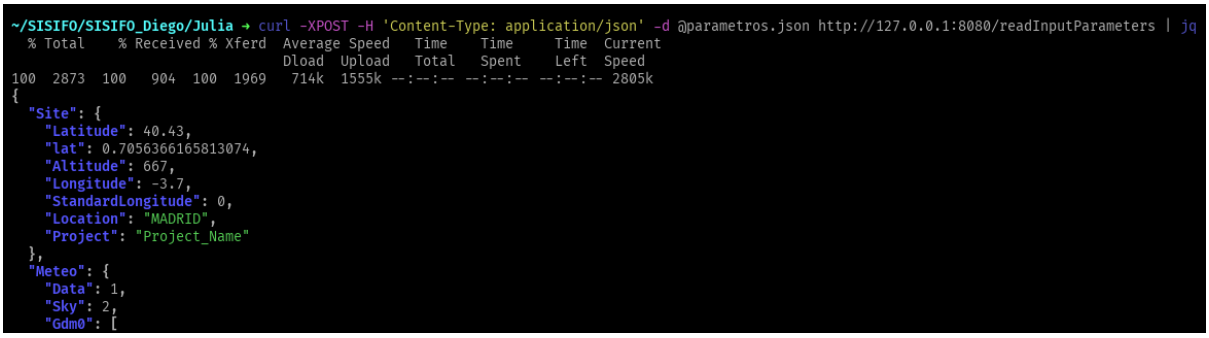
```
@post "/readInputParameters" function(req, InputParameters::Json{InputParameters})
    return InputParameters.payload
end

# start the web server
port = parse(Int, get(ENV, "APP_PORT", "8080"))
serve(port=port)
```

Captura 46. Servicio web Julia *(Fuente: Elaboración propia)*

Tras ejecutar esta función, queda un pequeño microservicio web escuchando en el puerto 8080, como se muestra en la captura 47:

Captura 47. Microservicio con Oxygen *(Fuente: Elaboración propia)*

Al realizar una petición HTTP a dicho servicio, haciendo una acción POST con el JSON anterior, la función devuelve el *payload.* En este caso se puede ver la salida de la orden cURL (Captura 48):



Captura 48. Petición HTTP y respuesta *(Fuente: Elaboración propia)*

El microservicio registra las llamadas a los endpoints en la salida por consola:

```
[ Info: 2024-12-01T21:23:25+0100 - 127.0.0.1:34734 - "POST /readInputParameters HTTP/1.1" 200
```

Tabla 2. Ejecución de readInputParameters como microservicio

	Inicio microsvc.	cURL 1ª ejec.	cURL 2ª ejec.
Tiempo de usuario	7.27s	1min 16.45s	0.00s
Tiempo de sistema	0.41s	0.96s	0.01s
Tiempo total	8.607s	1min 17.41s	0.009s

En la tabla 2 se indican los tiempos tras iniciar el microservicio (Precompilación de Oxygen) y las llamadas haciendo la orden cURL en una ejecución y sucesivas (Precompilación readInputParameters). Se advierte un elevado tiempo de compilación que varía entre 1 minuto y 1 minuto y 20 segundos, en las pruebas.

3.4.2 Creación de imagen de contenedor con microservicio

Como siguiente prueba de concepto, es necesario crear una imagen de contenedor que contenga el código del microservicio creado en el apartado anterior, además de Julia y las dependencias y paquetes necesarios.

Esta imagen de Docker debe tener lo necesario para correr Julia, siendo a su vez lo más minimalista posible, con el fin de reducir el tamaño lo máximo posible. Tener un mínimo de paquetes, librerías y software reduce además la exposición en el plano de la seguridad.

Para crear una imagen de Docker es posible hacer uso de un fichero Dockerfile con instrucciones nativas que permiten crear cada una de las capas que compondrán la imagen final. El Dockerfile para este microservicio Julia se muestra en la captura 49:

```dockerfile
FROM julia:1.9

# Set the working directory
WORKDIR /app

# Copy the current directory contents into the container at /app
COPY ./readInputParameters.jl /app/readInputParameters.jl

# Install dependencies
RUN julia -e 'using Pkg; Pkg.add(["Oxygen", "HTTP", "StructTypes"])'

# Expose the port the app runs on
EXPOSE 8080

# Run the application
CMD ["julia", "readInputParameters.jl"]
```

Captura 49. Dockerfile de readInputParameters *(Fuente: Elaboración propia)*

Quitando comentarios y líneas en blanco, en 6 líneas es posible, a partir de una imagen oficial del proyecto Julia en su versión 1.9: especificar el directorio de trabajo, copiar la función Julia creada en el apartado anterior, instalar las dependencias necesarias, exponer el puerto 8080 y especificar el comando que será ejecutado cuando el contenedor se ejecute.

Normalmente los proyectos de código abierto suben sus imágenes de contenedor a repositorios (*registries*) públicos. Gracias a estos repositorios es posible hacer uso de las imágenes referenciándolas por una etiqueta (tag), que suele coincidir con la versión a utilizar.

Uno de los repositorios de imágenes más utilizados es Docker Hub. Por norma general, cuando en una instrucción FROM no se especifica el repositorio de imágenes antes del nombre y etiqueta de la misma, se presupone el uso de Docker Hub.

En la captura 50 se puede apreciar el proceso de construcción de una imagen a partir del subcomando *docker build:*

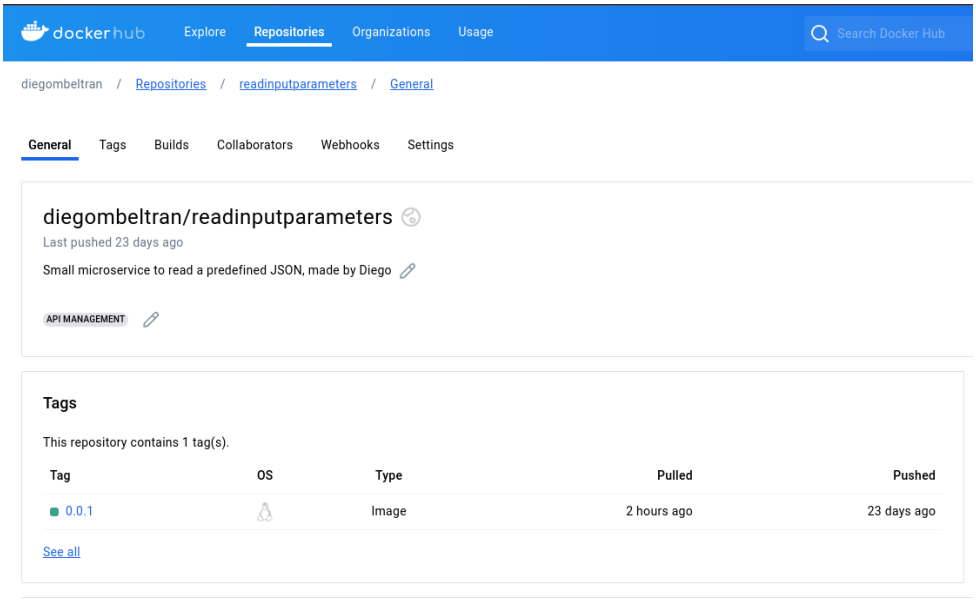
```
diegombeltran
in ~/SISIFO/SISIFO_Diego/Julia → docker build -t diegombeltran/readinputparameters:0.0.1 -f Dockerfile .
[+] Building 1.3s (10/10) FINISHED
 ⇒ [internal] load build definition from Dockerfile
 ⇒ ⇒ transferring dockerfile: 481B
 ⇒ [internal] load metadata for docker.io/library/julia:1.9
 ⇒ [auth] library/julia:pull token for registry-1.docker.io
 ⇒ [internal] load .dockerignore
 ⇒ ⇒ transferring context: 2B
 ⇒ [1/4] FROM docker.io/library/julia:1.9@sha256:829ddd53a80d14cfc02bae02198254ddef1de292c7cd6af694312fa8ad341290
 ⇒ [internal] load build context
 ⇒ ⇒ transferring context: 1.64kB
 ⇒ CACHED [2/4] WORKDIR /app
 ⇒ CACHED [3/4] COPY ./readInputParameters.jl /app/readInputParameters.jl
 ⇒ CACHED [4/4] RUN julia -e 'using Pkg; Pkg.add(["Oxygen", "HTTP", "StructTypes"])'
 ⇒ exporting to image
 ⇒ ⇒ exporting layers
 ⇒ ⇒ writing image sha256:f9629ccd4cbc3732870d8a12a72b82b921d9ef001155d12da4bdfd155e072eb5
 ⇒ ⇒ naming to docker.io/diegombeltran/readinputparameters:0.0.1
```

Captura 50. Creación de imagen Docker *(Fuente: Elaboración propia)*

Para el proyecto ha sido necesario crear un repositorio privado donde albergar las imágenes de los microservicios. Por simplicidad se ha escogido también Docker Hub. El nombre del repositorio es *diegombeltran*, siendo las imágenes públicas por simplificar la obtención de estas.

Una vez creado el repositorio privado y la imagen, se sube mediante el subcomando:

```
docker push diegombeltran/readinputparameters:0.0.1
```



Captura 51. Imagen readInputParameters en repositorio *(Fuente: Elaboración propia)*

En la captura 51 se muestra el repositorio con la imagen ya subida en Docker Hub.

Esta acción es necesaria llevarla a cabo para cada uno de los microservicios implicados en el proyecto.

3.4.3 Creación de servicio de Knative

Estando disponible la imagen de contenedor en un repositorio público, el siguiente paso es crear un servicio de Knative que haga uso de esta.

Knative Serving es un componente de la plataforma Knative que se enfoca en la ejecución de aplicaciones serverless. El componente serving permite crear servicios que escalan automáticamente según la demanda (Figura 52).

Una de las principales características es su capacidad para manejar el tráfico entrante de manera eficiente, incluyendo la gestión de versiones y revisiones de los servicios desplegados. Por ejemplo, es muy fácil llevar a cabo implementaciones *canary*, permitiendo dirigir una parte del tráfico hacia una nueva versión para probar su comportamiento antes de hacer un despliegue completo.

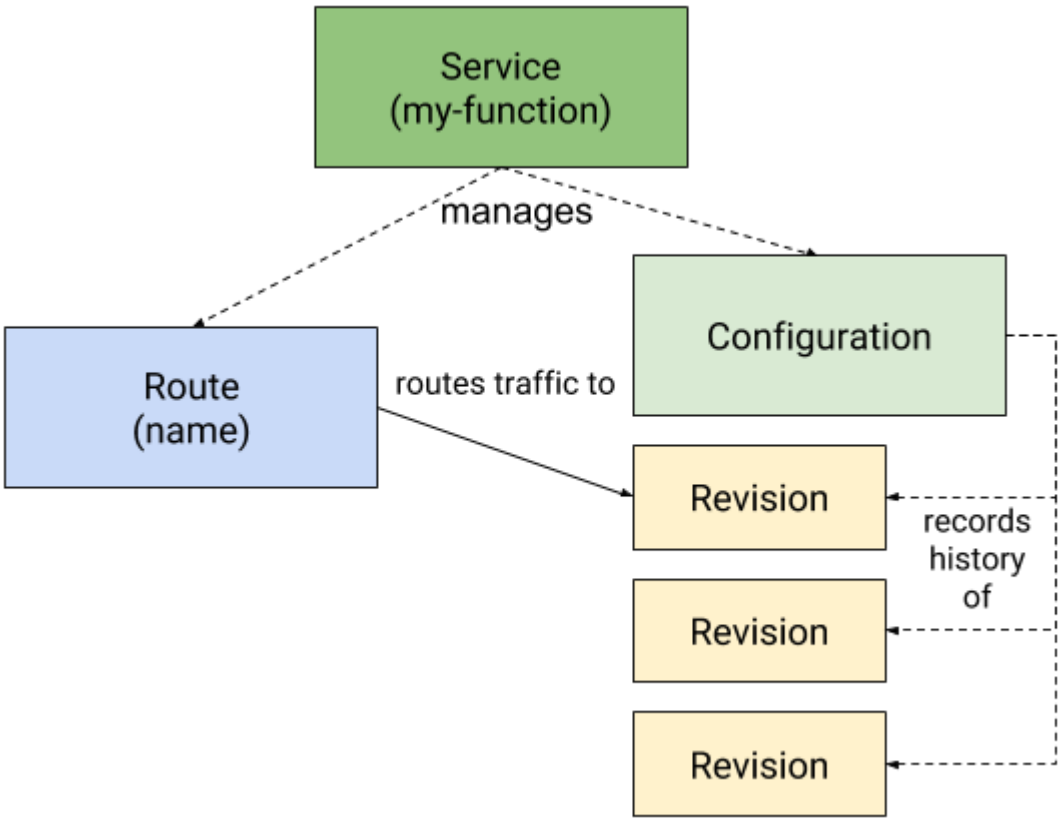


Figura 52. Arquitectura Knative Serving *(Fuente: Knative Project)*

Los principales recursos del componente serving son:

- Servicios: Gestiona automáticamente todo el ciclo de vida de la carga de trabajo. Controla la creación de otros objetos para garantizar que la aplicación tenga una ruta, una configuración y una nueva revisión para cada actualización del servicio. El servicio se puede definir para dirigir siempre el tráfico a la última revisión o a una revisión anclada.
- Rutas: Asigna un punto final de red a una o más revisiones. Se puede gestionar el tráfico de varias formas, incluyendo tráfico fraccionado y rutas con nombre.
- Configuraciones: Mantiene el estado deseado para el despliegue. Proporciona una separación limpia entre el código y la configuración. Modificar una configuración crea una nueva revisión.
- Revisiones: Es una instantánea en el tiempo del código y la configuración para cada modificación realizada en la carga de trabajo. Las revisiones son objetos inmutables y pueden conservarse tanto tiempo como sea útil.

La definición de una función o carga de trabajo de Knative Serving se realiza creando un objeto de Kubernetes del tipo *serving.knative.dev/v1*. Este objeto no es nativo en Kubernetes, y sólo estará disponible y será "interpretable" tras la instalación de Knative abordada en el apartado anterior correspondiente.

Es posible crear un fichero YAML con toda la configuración, como se muestra en la captura 53:

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: readinputparameters
  namespace: pruebadiego
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/min-scale: "1"  # Mínimo 0 instancias
        autoscaling.knative.dev/max-scale: "5"  # Máximo 5 instancias
    spec:
      containers:
        - image: diegombeltran/readinputparameters:0.0.1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: APP_PORT
              value: "8080"
```

Captura 53. Definición de servicio Knative *(Fuente: Elaboración propia)*

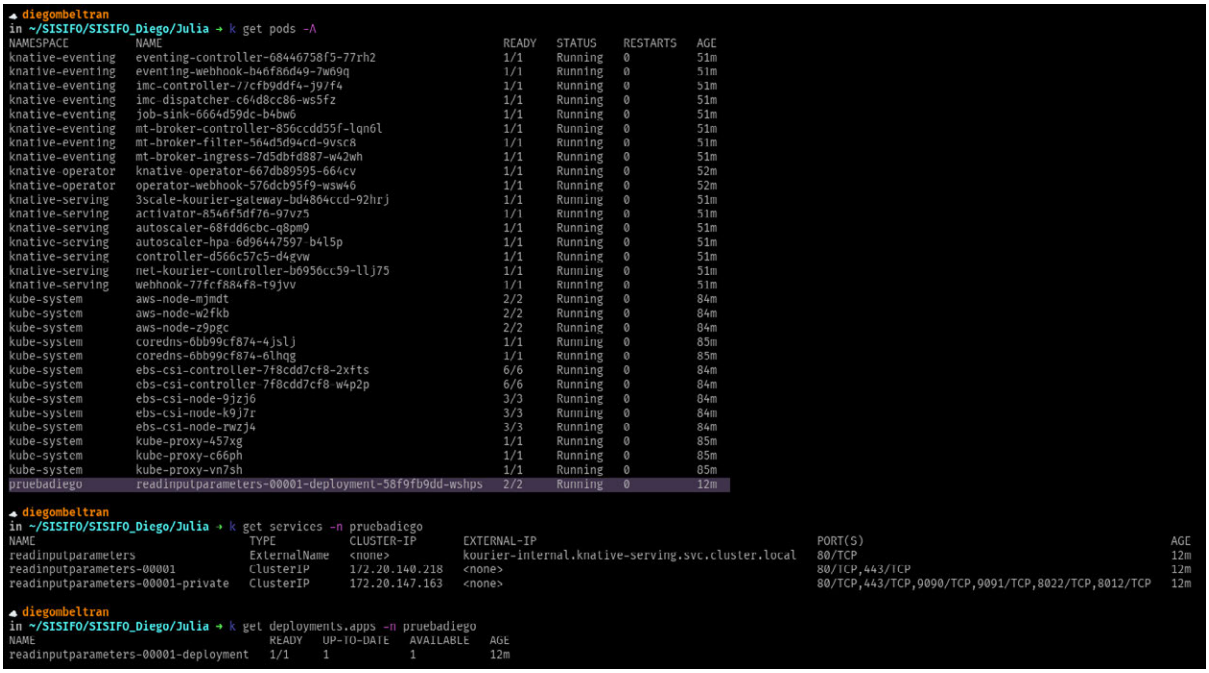
En este fichero se especifica el nombre de nuestro servicio de Knative, el espacio de nombres en el que se desplegará el pod asociado y una serie de ajustes concretos de Knative Serving.

En las especificaciones se indica la imagen de contenedor a utilizar, la política de descarga de la imagen, y el puerto expuesto.

En el apartado de anotaciones se especifica una instancia como mínimo corriendo en el clúster en todo momento, y un máximo de cinco, dependiendo del tráfico hacia el microservicio. En caso de establecer cero como número mínimo, el comportamiento esperado es que, en caso de no haber tráfico, la función no estará en ejecución.

Esto es especialmente interesante para reducir el coste, ya que los microservicios no tienen que estar siempre a la escucha y pueden ser invocados en caso de necesidad, a la vez que pueden pertenecer a una flota definible que puede aumentar o disminuir según la carga.

Al aplicar este manifiesto en el clúster, por debajo se crean varios recursos, siendo los más importantes los mostrados en la captura 54:



Captura 54. Recursos generados por manifiesto *(Fuente: Elaboración propia)*

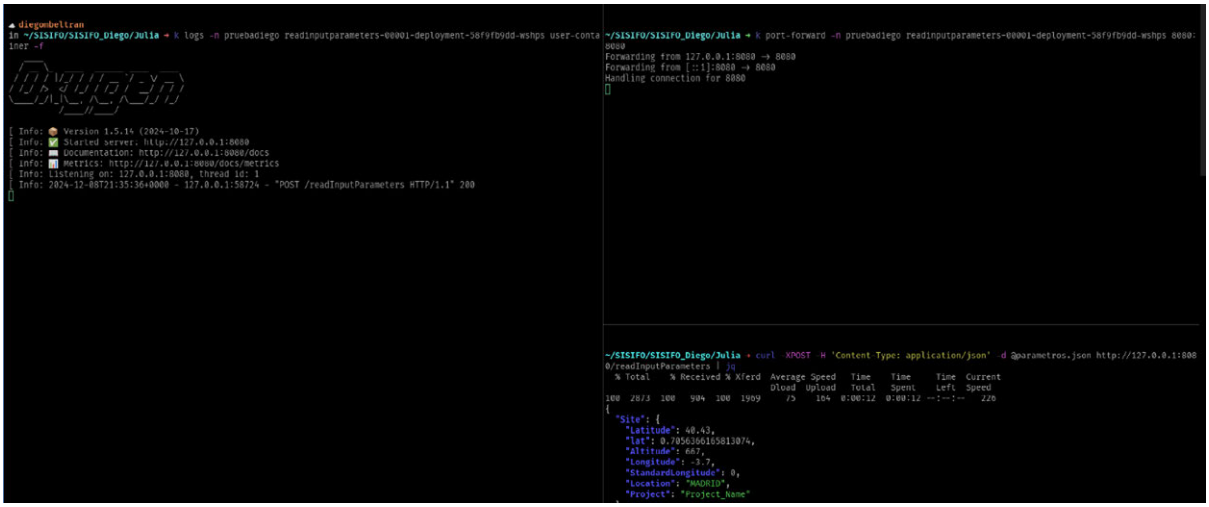
Cabe destacar que el pod surgido por este servicio de Knative contiene dos contenedores:

- **queue-proxy:** Añadido por Knative. Su función principal es gestionar diversos aspectos relacionados con el tráfico y el escalado automático de la aplicación.
- **user-container:** Es el contenedor original que hace uso de la imagen readInputParameters creada en el apartado anterior.

Si bien a estas alturas es posible realizar peticiones al microservicio a través de Internet, al subdominio proporcionado por el componente de red Kourier desplegado con Knative durante el proceso de instalación, se ha impedido esta acción por motivos de seguridad. El repartidor de carga tiene un agente de escucha en el puerto 443 (El puerto 80 ha sido desactivado por su naturaleza insegura), pero no cuenta con certificado TLS.

No obstante, no es necesario realizar la petición directamente al balanceador a través de Internet. Es posible utilizar el comando *kubectl port-forward*, que permite establecer un túnel para acceder a los recursos en un clúster desde una máquina local.

El comando *kubectl port-forward* se utiliza en Kubernetes para establecer un túnel que permite acceder a los recursos en un clúster desde la máquina local. Este comando facilita la depuración y el acceso a aplicaciones que se están ejecutando dentro del clúster, sin necesidad de exponerlas de manera pública. De esta manera no es necesario modificar la configuración ni del clúster ni del servicio, no se expone la aplicación a una red externa y proporciona un acceso temporal para pruebas y depuración sin cambios permanentes.



Captura 55. Ejecución de prueba *(Fuente: Elaboración propia)*

En la figura 55 pueden observarse tres terminales. En la de la izquierda se ejecuta el comando *kubectl logs* con el parámetro -f para obtener los logs de forma continua de una forma similar al comando tail de Unix. En la terminal superior derecha se ejecuta el comando *kubectl port-forward* para tunelizar la conexión desde la máquina local en el puerto 8080 hasta el pod en el puerto 8080 también. La terminal de abajo a la derecha es la misma petición HTTP mediante cURL que el apartado anterior.

Se puede observar en los logs como el JSON ha llegado al microservicio y ha sido serializado. El funcionamiento es el mismo que en ejecución local en el PC.

3.4.4 Comunicación de microservicios en Knative

El siguiente planteamiento consiste en el despliegue de dos funciones con Knative, además de un *broker.*

Como broker para esta prueba se ha escogido RabbitMQ. Es un proyecto de código abierto ampliamente utilizado para implementar sistemas de mensajería entre aplicaciones. Facilita la comunicación asincrónica y el intercambio de mensajes mediante el uso de protocolos de mensajería, como AMQP (Advanced Message Queuing Protocol).

El despliegue de RabbitMQ se ha realizado mediante un manifiesto de Kubernetes que contiene un objeto tipo Deployment y un objeto tipo Service. Es un despliegue muy sencillo no apto para entornos de producción.

Se pretende subir el JSON de parámetros de entrada al broker, al cual otro microservicio estará suscrito. Este microservicio realizará una petición al microservicio readInputParameters del apartado anterior. El esquema puede verse en la figura 56:

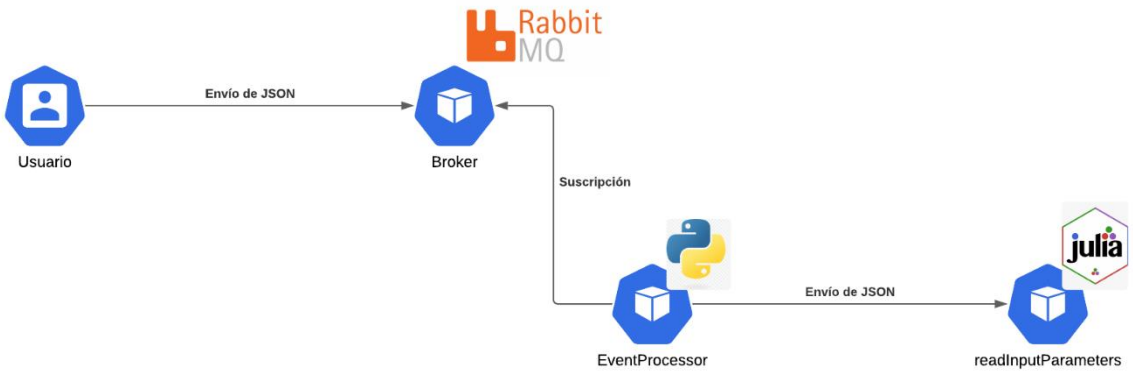


Figura 56. Esquema de prueba de concepto *(Fuente: Elaboración propia)*

Para poner en evidencia las ventajas y modularidad de una arquitectura basada en microservicios, el nuevo servicio llamado EventProcessor se ha desarrollado en Python.

En la captura 57 se encuentra la definición del nuevo servicio de Knative para el microservicio EventProcessor. Utiliza variables de entorno en el código, cuyos valores se han especificado en el manifiesto del servicio de Knative.

```yaml
apiVersion: serving.knative.dev/v1
kind: Service
metadata:
  name: rabbitmq-event-processor
  namespace: pruebadiego
spec:
  template:
    metadata:
      annotations:
        autoscaling.knative.dev/min-scale: "1"  # Mínimo 1 instancia
        autoscaling.knative.dev/max-scale: "5"  # Máximo 5 instancias
    spec:
      containers:
        - image: diegombeltran/rabbitmq_event_processor:0.0.1
          imagePullPolicy: Always
          ports:
            - containerPort: 8080
          env:
            - name: RABBITMQ_HOST
              value: "rabbitmq.rabbitmq.svc.cluster.local"
            - name: RABBITMQ_QUEUE
              value: "json_queue"
            - name: TARGET_URL
              value: "http://readinputparameters.pruebadiego.svc.cluster.local/readInputParameters"

```

Captura 57. Definición de servicio EventProcessor *(Fuente: Elaboración propia)*

Para referenciar al microservicio readInputParameters del apartado anterior, en la variable de entorno TARGET_URL se especifica el subdominio DNS interno de Kubernetes, compuesto por el nombre del servicio, el espacio de nombres seguido de svc; y por último el dominio local por defecto del clúster de Kubernetes: cluster.local.

Además, se ha probado a modificar el servicio de Knative del apartado anterior para que el mínimo de instancias corriendo sea de cero, con el objetivo de mostrar la función de autoescalado.

```
diegombeltran
in ~/SISIFO/SISIFO_Diego/Julia → python enviarjsonrabbitmq.py parametros.json
 [x] Sent {'PVGen': {'Struct': 1, 'PowNom_Total': 1000, 'PowNom_PerInverter': 100, 'PowNom_PerTransformer': 1000, 'PRV
PN': 1, 'PowReal_Total': 1000, 'NBGH': 10, 'NBGV': 10, 'NBT': 100, 'GroundRoof': {'Inclination_roof': 0, 'Orientation_
roof': 0, 'Inclination_generator': 35, 'Orientation_generator': 0, 'LNS': 1.5, 'AEO': 5, 'DEO': 0}}, 'Site': {'Latitud
e': 40.43, 'lat': 0.7056366165813074, 'Altitude': 667, 'Longitude': -3.7, 'StandardLongitude': 0, 'Location': 'MADRID'
, 'Project': 'Project_Name'}, 'Meteo': {'Data': 1, 'Sky': 2, 'Gdm0': [1980, 2680, 4430, 5080, 6480, 7210, 7320, 6430,
4970, 3350, 2130, 1600], 'TMm': [10.0, 12.7, 16.3, 17.8, 22.0, 29.0, 31.7, 31.2, 26.1, 20.2, 13.3, 10.0], 'Tmm': [3.0,
 3.1, 5.3, 6.2, 10.2, 15.1, 17.0, 16.5, 13.3, 10.4, 6.1, 3.4], 'Tlk': [3.0, 3.2, 2.9, 3.3, 3.4, 3.9, 3.8, 4.2, 3.9, 3.
9, 3.7, 3.1]}, 'PVMod': {'CellMaterial': 1, 'PowerModel': 1, 'CVPT': 0.5, 'NOCT': 48, 'P1000': None, 'P600': None, 'P2
00': None, 'LowGParam': None, 'AmorphousSi': {'SeasonalVariation': None, 'SeasonalPhase': None}}}
```

Captura 58. Envío de parámetros mediante script auxiliar *(Fuente: Elaboración propia)*

Como se puede ver en la captura 58, se hace uso de un pequeño script de ayuda escrito en Python que integra un cliente nativo de RabbitMQ. Este script es utilizado para enviar al broker RabbitMQ un mensaje con el JSON (parametros.json). Debido a que no hay acceso al clúster desde Internet por los motivos expuestos anteriormente, ha sido necesario ejecutar el comando *kubectl port-forward* de nuevo, esta vez tunelizando al puerto 5672 de RabbitMQ. El envío de parámetros mediante el script de ayuda se hace a 127.0.0.1:5672.

```
~/SISIFO/SISIFO_Diego/Julia → k get pods -n pruebadiego --watch
NAME                                                     READY   STATUS    RESTARTS   AGE
rabbitmq-event-processor-00001-deployment-5bb8f946bf-j9hxh   1/2     Running   0          75s
readinputparameters-00002-deployment-767647964d-sm8cd        0/2     Pending   0          0s
readinputparameters-00002-deployment-767647964d-sm8cd        0/2     Pending   0          0s
readinputparameters-00002-deployment-767647964d-sm8cd        0/2     ContainerCreating   0          0s
readinputparameters-00002-deployment-767647964d-sm8cd        1/2     Running             0          2s
readinputparameters-00002-deployment-767647964d-sm8cd        2/2     Running             0          20s
readinputparameters-00002-deployment-767647964d-sm8cd        2/2     Terminating         0          88s
readinputparameters-00002-deployment-767647964d-sm8cd        1/2     Terminating         0          110s
readinputparameters-00002-deployment-767647964d-sm8cd        0/2     Completed           0          118s
readinputparameters-00002-deployment-767647964d-sm8cd        0/2     Completed           0          119s
readinputparameters-00002-deployment-767647964d-sm8cd        0/2     Completed           0          119s
```

Captura 59. Ciclo de vida de servicio Knative *(Fuente: Elaboración propia)*

En la captura 59, se ha lanzado el comando *kubectl get pods* con el parámetro --watch. Este parámetro monitoriza el estado por el que pasan los pods y el tiempo relativo en el que va cambiando de estado. Se puede observar que readinputparameters-00002-deployment ha pasado de no existir a ser creado. Tras 60 segundos (Tiempo por defecto definido en Knative para autoescalar por inactividad), el pod es eliminado y el estado queda como completado.

Durante el tiempo de ejecución (running), se lanza el comando *kubectl logs* para comprobar si llega el mensaje desde el microservicio EventProcessor, y como se puede comprobar en la captura 60, llega una petición tipo POST.

```
diegombeltran
in ~/SISIFO/SISIFO_Diego/Julia → k logs -n pruebadiego readinputparameters-00002-deployment-767647964d-sm8cd
Defaulted container "user-container" out of: user-container, queue-proxy
   ____
  / __ \_ ___  ____ _____ ___  ___
 / / / / |/_/ / / / __ `/ _ \/ __ \
/ /_/ />  </ /_/ / /_/ /  __/ / / /
\____/_/|_|\__, /\__,_/\___/_/ /_/
          /____//____/

[ Info: 📦 Version 1.5.14 (2024-10-17)
[ Info: ✅ Started server: http://127.0.0.1:8080
[ Info: 📖 Documentation: http://127.0.0.1:8080/docs
[ Info: 📊 Metrics: http://127.0.0.1:8080/docs/metrics
[ Info: Listening on: 127.0.0.1:8080, thread id: 1
[ Info: 2024-12-08T22:29:31+0000 - 127.0.0.1:50434 - "POST /readInputParameters HTTP/1.1" 200
```

Captura 60. Recepción de petición tras autoescalado *(Fuente: Elaboración propia)*

3.4.5 Implementación de MainProgram e interfaz para resultados

El diagrama de arquitectura propuesto para esta fase es el mostrado a continuación (Figura 61):

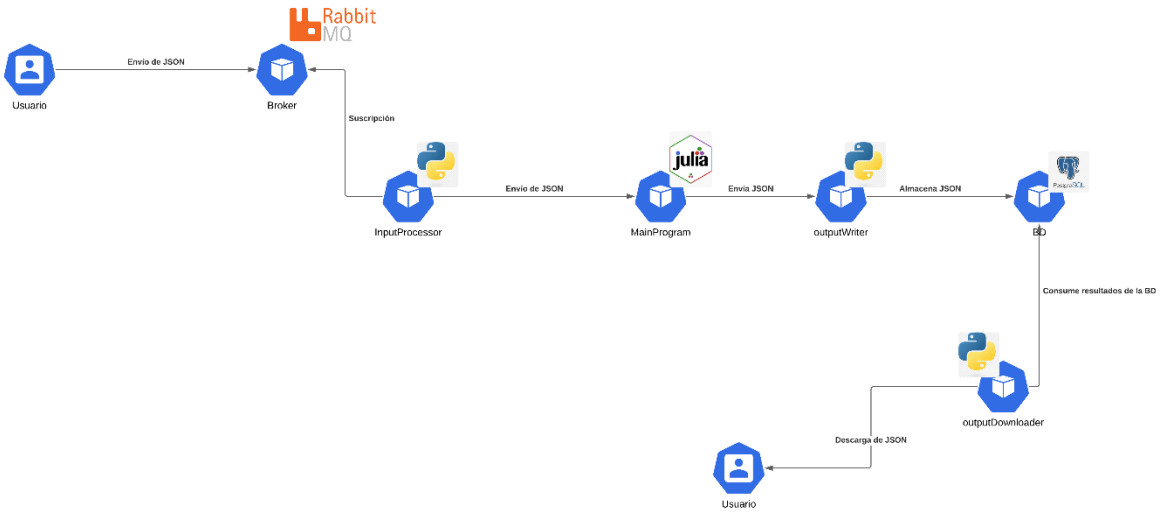


Figura 61. Diagrama MainProgram Web *(Fuente: Elaboración propia)*

Como primera acción se pretende adaptar MainProgram como un servicio web Julia para que reciba un fichero JSON de la cola de mensajes de RabbitMQ (haciendo de intermediario el servicio inputProcessor), se serialice el contenido, se procese, y se vuelva a de-serializar para su envío vía JSON a un tercer servicio llamado outputWriter.

Este nuevo servicio recibirá el resultado de la simulación y lo almacenará en una base de datos PostgreSQL. La base de datos es consumida por un último servicio llamado outputDownloader, que hace a su vez de interfaz web para el usuario, dónde éste puede descargar los resultados de las simulaciones.

El uso de las funciones nativas de Oxygen.jl para la serialización y de-serialización de objetos con JSON requiere el uso de *Structs*[1] en Julia (Captura 62). Si bien es posible hacer uso de *dicts* (Similares a los de Python o PHP) realizando el procesado de datos de forma manual mediante otras librerías como JSON, se decide utilizar *structs*, dado que se conoce la estructura de los datos con antelación y el uso de esta característica del lenguaje es más eficiente en términos de memoria y rendimiento, además de proporcionar seguridad de tipo.

[1] Un *struct* es una estructura de datos que permite definir tipos compuestos agrupando múltiples campos, cada uno con su propio nombre y tipo.

```
struct InputParameters
    SITE::SITE
    METEO::METEO
    PVMOD::PVMOD
    PVGEN::PVGEN
    BOS::BOS
    OPTIONS::OPTIONS
    TIME::TIME
    ECO::ECO
end

StructTypes.StructType(::Type{InputParameters}) = StructTypes.Struct()
```

Captura 62. Ejemplo de struct *(Fuente: Elaboración propia)*

Tras una reescritura, el código pasa a no usar diccionarios y hacer uso de estructuras en la mayoría de las funciones que hacen uso de parámetros de entrada y son susceptibles de serializar desde un JSON. Se aprovecha para solventar algunos problemas de consistencia y condiciones de carrera. Un ejemplo es la inicialización de *arrays* vacíos con ceros.

Por último, se crea una ruta. Al hacer una llamada HTTP tipo POST a "/runSimulation", se ejecuta la función MainProgram con los parámetros de entrada serializados y se envía el resultado mediante otra petición HTTP tipo POST al siguiente microservicio de la cadena, outputWriter. La URL del microservicio se obtiene de variable de entorno, siendo el valor por defecto el mostrado en la captura 63:

```
@post "/runSimulation" function (req, InputParameters::Json{InputParameters})
    site = SISIFOLibrary.SITE(InputParameters.payload.SITE.Latitude, InputParameters.pa
    meteo = SISIFOLibrary.METEO(InputParameters.payload.METEO.Data, InputParameters.pay
    pvmod = SISIFOLibrary.PVMOD(InputParameters.payload.PVMOD.CellMaterial, InputParame
    pvgen = SISIFOLibrary.PVGEN(InputParameters.payload.PVGEN.Struct, InputParameters.p
    bos = SISIFOLibrary.BOS(SISIFOLibrary.INVERTER(InputParameters.payload.BOS.INVERTEF
    options = SISIFOLibrary.OPTIONS(InputParameters.payload.OPTIONS.Application, InputF
    time = SISIFOLibrary.TIME(InputParameters.payload.TIME.LocalTime, InputParameters.p
    eco = SISIFOLibrary.ECO(InputParameters.payload.ECO.LoanPerc, InputParameters.paylo


    output = MainProgram(site, meteo, pvmod, pvgen, bos, options, time, eco)
    json_output = JSON.json(output)
    #return json_output

    url = get(ENV, "WRITER_ENDPOINT", "http://sisifo-outputwriter.sisifo.svc/write")
    response = HTTP.post(url, [("Content-Type", "application/json")], json_output)

    return response
```

Captura 63. Ruta @post de Oxygen *(Fuente: Elaboración propia)*

Como último paso, se empaqueta la función como imagen de contenedor mediante otro Dockerfile. La única particularidad en este caso es la adición de SISIFOLibrary.jl como dependencia de MainProgram. Esta librería contiene todas las funciones y definición de estructuras adicionales. Respecto a la definición del servicio de Knative, el único cambio realizado son las variables de entorno para definir puerto de escucha del servicio, así como URL destino del microservicio outputWriter.

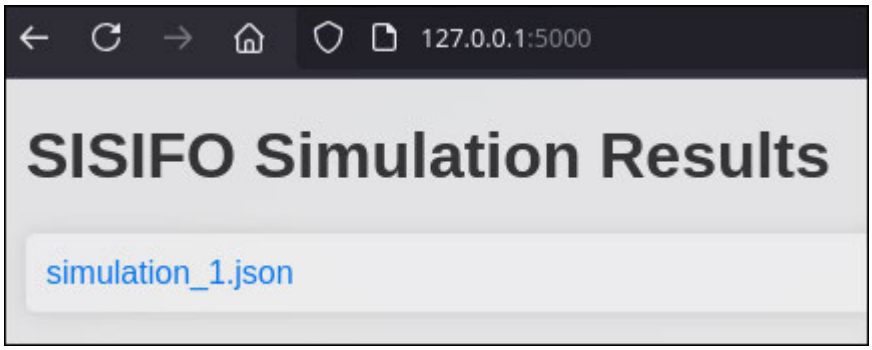
Los dos siguientes microservicios, outputWriter y outputDownloader, están escritos en Python, por el extenso número de paquetes y utilidades para operar con el código, en este caso PostgreSQL (Captura 64).

```python
from flask import Flask, request, jsonify
import psycopg2
import json
import os

app = Flask(__name__)

db_host = os.getenv('POSTGRES_HOST', 'localhost')
db_name = os.getenv('POSTGRES_DB', 'outputs')
db_user = os.getenv('POSTGRES_USER', 'user')
db_password = os.getenv('POSTGRES_PASSWORD', 'password')

def init_db():
    conn = psycopg2.connect(
        host=db_host,
        database=db_name,
        user=db_user,
        password=db_password
    )
    cursor = conn.cursor()
    cursor.execute('''
        CREATE TABLE IF NOT EXISTS results (
            id SERIAL PRIMARY KEY,
            data TEXT NOT NULL
        )
    ''')
    conn.commit()
    cursor.close()
    conn.close()

def save_to_db(message):
    conn = psycopg2.connect(
        host=db_host,
```

Captura 64. Cliente PostgreSQL para Python *(Fuente: Elaboración propia)*

Detallando outputWriter, el código recibe el JSON resultado de MainProgram (En una ruta /write) y lo vuelca a una tabla de una base de datos corriendo en el clúster. Es por tanto obligatorio crear un manifiesto de Kubernetes con un despliegue muy sencillo y no apto para producción de PostgreSQL, teniendo en cuenta que cualquier motor SQL puede servir para esta acción. De hecho, durante las pruebas locales iniciales, se ha hecho uso de SQLite.

Por otra parte, outputDownloader también tiene un cliente de PostgreSQL nativo para consumir los resultados de la tabla. Mediante dos rutas, es posible listar los resultados con una pequeña página web con HTML/CSS y descargarlos en JSON (Captura 65):



Captura 65. Interfaz web con Flask *(Fuente: Elaboración propia)*

Esta fase del proyecto ha sido una de las que más tiempo de desarrollo y depuración han tomado, siendo necesario comparar con detalle las salidas una a una de las funciones de MainProgram: mediante ejecución local, ejecución vía microservicios también en local, y por último en el propio clúster de Kubernetes.

Tabla 3. Ejecución de MainProgram como microservicio

	Inicio microsvc.	cURL 1ª ejec.	cURL 2ª ejec.
Tiempo de usuario	7.27s	1min 11.17s	0.92s
Tiempo de sistema	0.41s	0.86s	0.01s
Tiempo total	7.68s	1min 12.03s	0.93s

Una vez obtenido el mismo resultado en varias ejecuciones de la simulación con unos mismos parámetros de entrada se da por finalizada la fase.

3.4.6 Funciones de MainProgram como microservicios

Hasta este momento el código adaptado de esta versión de SISIFO no es enteramente basado en microservicios. Hay una función, MainProgram que sí lo es y hace uso de *serverless*, pero la organización actual del código es monolítica. La última fase del proyecto consiste en el desacoplamiento de dos funciones invocadas dentro de MainProgram a microservicios independientes. Se han escogido dos a modo representativo: Eccentricity y HorizontalIrradiances, del total de las que hay. La primera es una función más simple con parámetros de entrada y salida más reducidos; mientras que la segunda es de gran tamaño y a su vez hace uso de funciones internas adicionales y parámetros de gran tamaño.

La primera función consta de un parámetro de entrada tipo TIME, definido como struct previamente. Tiene una ruta para peticiones HTTP tipo POST en "/calculate".

Durante el flujo del programa, una vez invocado el servicio MainProgram, se ha modificado el código para que en vez de hacer una llamada local a la librería a la función Eccentricity, se realice una petición HTTP a este nuevo microservicio. El parámetro de entrada es de-serializado a formato JSON y se pasa como parámetro (Captura 66):

```
# Eccentricity correction factor, dimensionless
eccentricity_url = get(ENV, "ECCENTRICITY_URL", "http://sisifo-eccentricity.sisifo.svc/calculate")
eccentricity_response = HTTP.post(eccentricity_url, [("Content-Type", "application/json")], JSON.json(TIME))
SUNMOT_Eo = JSON.parse(String(eccentricity_response.body))

# Solar declination, radian
SUNMOT_delta = SolarDeclination(TIME)
```

Captura 66. Llamada a Eccentricity *(Fuente: Elaboración propia)*

El microservicio Eccentricity, interpreta los datos JSON recibidos y vuelve a serializar en un objeto struct tipo TIME.

Una vez se recibe respuesta, MainProgram continúa su ejecución con el resultado de la función como valor de la variable *SUNMOT_Eo.* Cabe destacar que esta es una operación síncrona. Hasta que la petición al microservicio Eccentricity no se resuelve, MainProgram no finaliza su ejecución.

Para la segunda función, HorizontalIrradiances, se ha escogido un método de actuación diferente. En vez de deserializar los parámetros de entrada, se ha optado por el uso de una solución de caché, siendo Memcached la opción escogida. Se desea evaluar la diferencia de rendimiento entre ambas soluciones.

Una de las ventajas de utilizar Memcached en esta arquitectura de microservicios es que permite almacenar objetos directamente en memoria, lo que puede minimizar la necesidad de serializar y deserializar datos constantemente. Al almacenar los datos en su forma nativa,

se elimina el costo computacional y el tiempo asociados con estos procesos, acelerando así el acceso y manipulación de la información. Esto es especialmente beneficioso en entornos donde cada milisegundo cuenta, ya que la serialización y deserialización pueden introducir latencia.

Otra de las ventajas de usar una solución de caché en memoria como esta es salvar los problemas derivados de peticiones entre microservicios de gran tamaño. Los proveedores de funciones como servicio (FaaS), suelen imponer límites en el tamaño de las peticiones y respuestas para garantizar su desempeño e incluso la seguridad ante DDoS (D*istributed Denial-of-Service*). Estos límites abordan el tamaño máximo del *payload* que una función puede recibir o devolver en una única ejecución. En AWS Lambda, por ejemplo, el límite de tamaño de la solicitud de entrada es de 6 MB (para eventos asíncronos) y el payload de las respuestas puede ser de hasta 6 MB también.

En el caso de Knative es posible hacer uso del parámetro *data-max-size* para especificar este comportamiento, aunque durante el proyecto no se ha hecho uso de ello.

La arquitectura de Memcached en el contexto de la aplicación se muestra en la figura 67:

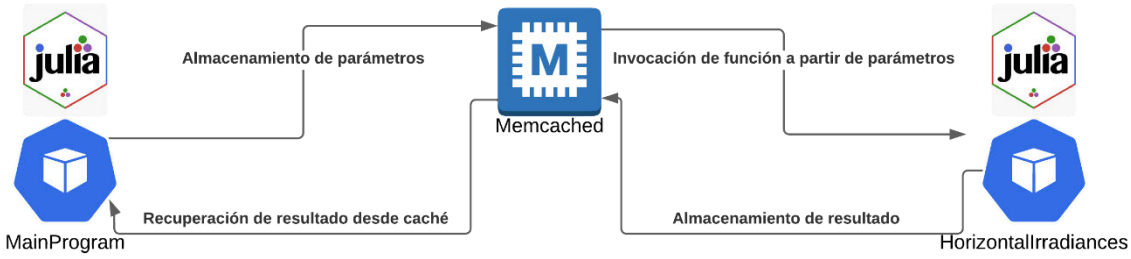


Figura 67. Detalle de comunicación con Memcached *(Fuente: Elaboración propia)*

Una de las ventajas es la no deserialización a JSON y posterior tratamiento de este mediante conversión a objetos binarios. Al utilizar objetos Julia directamente en memoria la velocidad es mayor, especialmente con matrices o contenido de gran tamaño.

La implementación de este modelo en el código consiste en crear sendos clientes tanto en MainProgram como en Horizontalrradiances. En la captura 68 se dispone un ejemplo del código en MainProgram:

```
# Calculation of horizontal irradiances
memcached_url = get(ENV, "MEMCACHED_URL", "memcached.memcached.svc")
mc = MemcacheClient(memcached_url, 11211)

set(mc, "hi_parameter", hiParameter(SUNMOT, SUNPOS, AMI, SITE, METEO, OPTIONS, TIME))

hi_url = get(ENV, "HI_URL", "http://sisifo-horizontalirradiances.sisifo.svc/calculate")
hi_response = HTTP.get(hi_url)

if hi_response.status == 200
    HI = get(mc, "hi_result")
else
    error("Something went wrong with HorizontalIrradiances microservice")
end
```

Captura 68. Creación de cliente Memcached *(Fuente: Elaboración propia)*

Tras las pruebas locales, se comprueba que el resultado de la simulación con los parámetros de entrada es la misma que en anteriores escenarios, por lo que se empaquetan las funciones como imágenes de contenedor y se despliegan en su totalidad en el clúster de Kubernetes, dando por finalizada la fase de implementación.

Como último apunte, se realiza un cambio en la parte final de la arquitectura. Tras las primeras pruebas y ejecuciones en el clúster se localiza un problema en el el procesamiento de los parámetros en JSON del microservicio outputWriter en la función Python *JSON.dumps*.

Con el fin de simplificar y evitar errores en componentes, se elimina la base de datos PostgreSQL y el microservicio outputDownloader. Se deja el microservicio outputWriter como interfaz web de usuario y los resultados JSON de la simulación se almacenan localmente en formato JSON para su descarga.

4 Resultados

El diagrama final de la arquitectura es el mostrado en la figura 69:

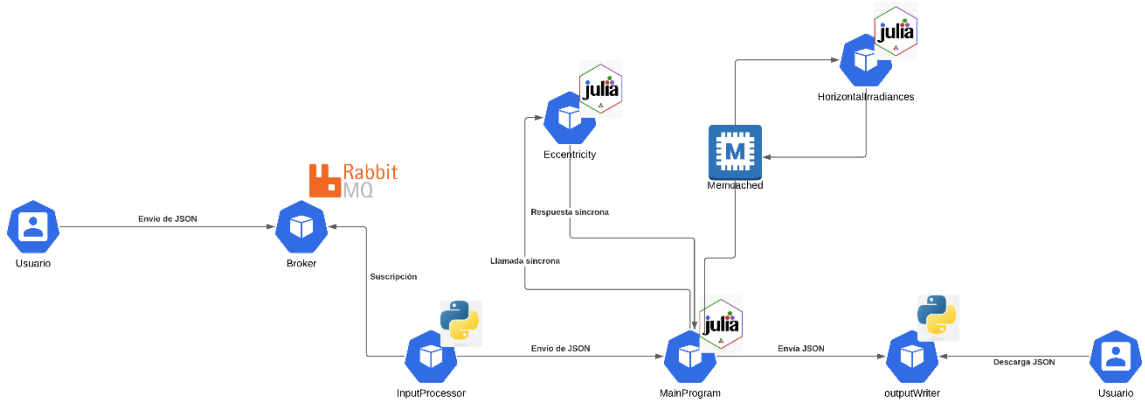
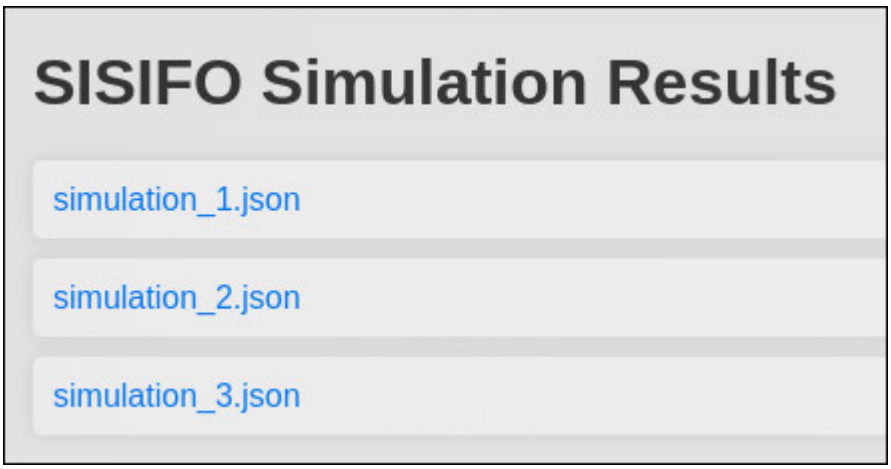


Figura 69. Diagrama de arquitectura final *(Fuente: Elaboración propia)*

Este fragmento de la aplicación original es ahora una arquitectura basada en microservicios y diferentes herramientas estándar en la industria. Se ejecuta en un clúster de Kubernetes con Knative como framework FaaS y es completamente escalable en todos y cada uno de sus componentes (Captura 70). Todo el entorno es perfectamente reproducible de forma automática y programática. El código se encuentra en repositorios de GitHub.

```
in SISIFO_Julia → k get pods -A
NAMESPACE              NAME                                                        READY   STATUS    RESTARTS   AGE
knative-eventing       eventing-controller-68446758f5-kkzbc                        1/1     Running   0          39m
knative-eventing       eventing-webhook-b46f86d49-6fl2c                            1/1     Running   0          39m
knative-eventing       imc-controller-77cfb9ddf4-djmlw                             1/1     Running   0          39m
knative-eventing       imc-dispatcher-c64d8cc86-65f6z                              1/1     Running   0          39m
knative-eventing       job-sink-6664d59dc-g5j4j                                    1/1     Running   0          39m
knative-eventing       mt-broker-controller-856ccdd55f-vpzs9                       1/1     Running   0          39m
knative-eventing       mt-broker-filter-564d5d94cd-b7whd                           1/1     Running   0          39m
knative-eventing       mt-broker-ingress-7d5dbfd887-mrn2t                          1/1     Running   0          39m
knative-operator       knative-operator-667db89595-pvw7l                           1/1     Running   0          40m
knative-operator       operator-webhook-576dcb95f9-2dmns                           1/1     Running   0          40m
knative-serving        3scale-kourier-gateway-bd4864ccd-5xknw                      1/1     Running   0          39m
knative-serving        activator-8546f5df76-f22t9                                  1/1     Running   0          39m
knative-serving        autoscaler-68fdd6cbc-ldvvm                                  1/1     Running   0          39m
knative-serving        autoscaler-hpa-6d96447597-tgn8x                             1/1     Running   0          39m
knative-serving        controller-d566c57c5-hvrgv                                  1/1     Running   0          39m
knative-serving        net-kourier-controller-b6956cc59-72c85                      1/1     Running   0          39m
knative-serving        webhook-77fcf884f8-tbpb5                                    1/1     Running   0          39m
kube-system            aws-node-d5x89                                              2/2     Running   0          42m
kube-system            aws-node-f6fnq                                              2/2     Running   0          41m
kube-system            aws-node-n6sdb                                              2/2     Running   0          42m
kube-system            coredns-6bb99cf874-glgnw                                    1/1     Running   0          42m
kube-system            coredns-6bb99cf874-s9zs6                                    1/1     Running   0          42m
kube-system            ebs-csi-controller-7f8cdd7cf8-kwhsk                         6/6     Running   0          42m
kube-system            ebs-csi-controller-7f8cdd7cf8-zgszb                         6/6     Running   0          42m
kube-system            ebs-csi-node-5zb5n                                          3/3     Running   0          42m
kube-system            ebs-csi-node-fl2qt                                          3/3     Running   0          42m
kube-system            ebs-csi-node-mft6w                                          3/3     Running   0          42m
kube-system            kube-proxy-56l5w                                            1/1     Running   0          42m
kube-system            kube-proxy-fpch9                                            1/1     Running   0          42m
kube-system            kube-proxy-xk8lf                                            1/1     Running   0          42m
memcached              memcached-7fbcc6b75d-p57sj                                  1/1     Running   0          34m
rabbitmq               rabbitmq-6c66d66c6-qqh7c                                    1/1     Running   0          26m
sisifo                 sisifo-eccentricity-00001-deployment-57bc766c86-zvftj       2/2     Running   0          34m
sisifo                 sisifo-horizontalirradiances-00001-deployment-6f7b66f649-vz7lc  2/2     Running   0          34m
sisifo                 sisifo-inputprocessor-00001-deployment-58fbcd9765-ckrrr     2/2     Running   0          20m
sisifo                 sisifo-mainprogram-00001-deployment-77c76cdc5c-tb2td        2/2     Running   0          34m
sisifo                 sisifo-outputwriter-00001-deployment-c675796b5-r75w5        2/2     Running   0          34m
```

Captura 70. Componentes desplegados *(Fuente: Elaboración propia)*



Captura 71. Interfaz web final *(Fuente: Elaboración propia)*

Es posible acceder al microservicio outputWriter mediante port-forward para descargar los resultados (Captura 71).

Probablemente los resultados más relevantes para tener en cuenta son los tiempos llevados a cabo por cada una de las fases del proyecto.

4.1 Tiempo de despliegue

Se procede a medir el tiempo en el que es posible tener en funcionamiento esta aplicación en un proveedor cloud y una simulación realizada.

Para toda la infraestructura se puede hacer de forma programática con Terraform como se describe en el apartado de desarrollo.

Ejecutando el comando *time terraform apply* y esperando a que toda la infraestructura se levante se obtiene el siguiente resultado:

```
terraform apply -refresh=true -var-file=environments/diego.tfvars  9,44s user 1,09s system 1% cpu 13:28,06 total
```

El siguiente paso es el despliegue de los componentes de Knative. Se ha creado un pequeño script en bash para desplegar los manifiestos de todos los componentes en un solo paso.

```
./deploy.sh  2,70s user 0,41s system 19% cpu 15,602 total
```

Una vez ejecutado el script, se comprueba el tiempo tomado en desplegar los pods necesarios para funcionar (Incluye la descarga desde los repositorios de imágenes de contenedor desde la infraestructura de AWS, por lo que este tiempo puede variar ligeramente)

Según el último cambio de estado del último pod a RUNNING, se obtienen 52s.

Por último, se hace lo mismo con los manifiestos de los componentes de la aplicación, tanto los servicios de Knative para cada función, como el broker y el caché.

```
./deploy.sh  1,82s user 0,27s system 41% cpu 4,976 total
```

Siendo el cambio de estado a RUNNING del último pod un tiempo total de 3:12s. El tiempo en descargar las imágenes y levantar todos los pods es de menos de 40 segundos, pero el proceso de precompilación de las funciones Julia (la parte del framework Oxygen) hace que tarde un poco más en estar READY y por tanto que el pod se considere RUNNING.

En la tabla 2 se detallan los tiempos para el despliegue de la arquitectura desde cero:

Tabla 4. Tiempos de despliegue

ACCIÓN	TIEMPO
Despliegue de infraestructura	13 minutos y 28 segundos
Despliegue de Knative	1 minuto y 7.6 segundos
Despliegue de aplicación	3 minutos y 16.97 segundos
TIEMPO TOTAL	17 minutos y 52.57 segundos

4.2 Comparativa de tiempos de funciones Julia

En la tabla se muestra una comparativa de tiempos al ejecutar las funciones, tanto local como microservicios.

Tabla 5. Comparación de tiempos en Julia

MainProgram (Ejecución Local)			
	PHP	Julia 1ª ejec.	Julia 2ª ejec.
Tiempo de usuario	0.41s	8.35s	0.9s
Tiempo de sistema	0.07s	0.21s	0.07s
Tiempo total	0.479s	8.547s	0.97s
readInputParameters (Ejecución como microservicio)			
	Inicio microsvc.	cURL 1ª ejec.	cURL 2ª ejec.
Tiempo de usuario	7.27s	1min 16.45s	0.00s
Tiempo de sistema	0.41s	0.96s	0.01s
Tiempo total	8.607s	1min 17.41s	0.009s
MainProgram (Ejecución como microservicio			
	Inicio microsvc.	cURL 1ª ejec.	cURL 2ª ejec.
Tiempo de usuario	7.27s	1min 11.17s	0.92s
Tiempo de sistema	0.41s	0.86s	0.01s
Tiempo total	7.68s	1min 12.03s	0.93s

Los tiempos de inicio de cada microservicio consisten en el tiempo de precompilación de Oxygen tras la primera ejecución. El servicio queda escuchando en el puerto 8080. Al enviar los parámetros en JSON mediante cURL, se precompilan el resto de las funciones y la librería. En segunda y sucesivas peticiones el tiempo de respuesta es mucho menor.

4.3 Tiempo de ejecución de simulación total

Para esta fase se mide el tiempo que tarda en estar disponible el resultado de una simulación desde que se envían los parámetros en JSON al broker. Para ello se mide el tiempo en el que aparece un nuevo fichero JSON en el directorio "uploads" del microservicio outputWriter. Tras cronometrar el tiempo, en una primera ejecución con todas las funciones recién desplegadas, se obtiene un tiempo de 32.33s.

Tabla 6. Tiempos de ejecución

ACCIÓN	TIEMPO
Primera ejecución	32.33 segundos
Segunda ejecución y sucesivas	1 segundo

Sucesivas ejecuciones tienen un tiempo inferior o cercano al segundo en todos los casos. Esto es debido al tiempo de precompilación de los tres microservicios Julia. En caso de reiniciar uno de los pods de las funciones y dejar los otros dos, el tiempo de precompilación de las dos funciones restantes sin reiniciar no se suma a la hora de precompilar la reiniciada.

4.4 Comparativa peticiones HTTP vs Memcached

Para comparar la eficiencia por el uso de Memcached frente a HTTP con JSON, se han usado dos versiones diferentes de MainProgram y HorizontalIrradiances, haciendo uso de Oxygen y JSON; y haciendo uso de un cliente de Memcached.

Tras realizar una primera ejecución para evitar el *handicap* del proceso de precompilación, se modifica el código de MainProgram para hacer uso del paquete *Dates* de Julia y poder medir de alguna manera el tiempo, como se ve en la figura 72:

```
# Calculation of horizontal irradiances
start_time = now()
memcached_url = get(ENV, "MEMCACHED_URL", "memcached.memcached.svc")
mc = MemcacheClient(memcached_url, 11211)
end_time = now()
response_time = end_time - start_time
response_time = Dates.value(response_time) / 1000
```

Captura 72. Medición de tiempos *(Fuente: Elaboración propia)*

Para ambos escenarios, se envían alrededor de 20 peticiones consecutivas y se miden los tiempos de respuesta. En las pruebas HTTP, el tiempo promedio de respuesta es de 60ms, mientras que las consultas a Memcached mantienen un promedio de 15ms. Teniendo en cuenta que todos los microservicios corren en una red privada virtual en la nube de muy baja latencia, los resultados muestran una reducción significativa.

Tabla 7. Tiempos HTTP vs Memcached

ACCIÓN	TIEMPO
HorizontalIrradiances vía petición HTTP	60ms
HorizontalIrradiances vía Memcached	15ms

5 Presupuesto

Se detalla a continuación el presupuesto estimado para el desarrollo del proyecto, considerando los siguientes factores:

El costo de los servicios de computación de AWS se beneficia de una capa gratuita, en la cual no se generan cargos hasta sobrepasar un cierto número de horas de uso. Durante el desarrollo, se ha creado la infraestructura junto con todos sus componentes bajo demanda, eliminándola al terminar cada sesión de trabajo. Durante la fase de adaptación del código fuente a Julia, no fue necesario mantener esta infraestructura operativa.

A continuación, se presentan los precios por hora de los componentes que han generado mayor gasto, excluyendo aquellos que son gratuitos (como el tráfico saliente desde la plataforma de AWS hasta cierto nivel de uso, KMS y la monitorización de CloudWatch). También se incluye una estimación del costo para mantener el funcionamiento 24/7 durante un mes.

Finalmente, se proporciona un presupuesto para un equipo pequeño compuesto por un arquitecto de software senior, un desarrollador de software junior y un administrador de sistemas junior. Este presupuesto asume 40 horas de trabajo dedicadas a la implementación del proyecto, sin considerar la documentación previa y asumiendo que los miembros del equipo tienen experiencia demostrada en las tecnologías empleadas.

Es importante señalar que el cálculo de los costes no incluye el 21% de IVA.

Servicios de AWS

Tabla 8. Gasto en AWS

CONCEPTO	DESCRIPCIÓN	COSTE POR HORA	COSTE MENSUAL
Plano de control EKS	Gestión del plano de control de Kubernetes en EKS	$0.10	$33.29
Instancias EC2	Tres nodos (instancias EC2) tipo t3.medium	$0,0456	$73.00
Balanceadores de Carga (ELB)	Tres balanceadores de carga por cada zona de disponibilidad	$0.0252 + $0.008/lcu[2]	$55.20
Tráfico NAT Gateway	Tráfico procesado	$0.048/h + $0.048/gb	$35 (Estimado)
TOTAL	Total, sumado	$0.2748	$161.49

[2] Una LCU (Load Balancer Capacity Units) mide una serie de dimensiones en las que el balanceador de carga de aplicaciones procesa el tráfico. Tiene cuatro dimensiones, entre las que se encuentran el número de nuevas conexiones, las conexiones activas, los bytes procesados y las evaluaciones de reglas. Se cobra por la dimensión de más uso. En este proyecto se desestima este coste porque no alcanza el mínimo para facturar.

Servicios de consultoría

Tabla 9. Gasto hipotético de trabajadores

CONCEPTO	COSTE POR HORA	HORAS	COSTE TOTAL
Arquitecto de software senior	60€	40	2400€
Ingeniero de software junior	35€	40	1400€
Administrador de sistemas junior	35€	40	1400€
TOTAL	130€	40	5200€

6 Conclusiones

Este proyecto ha representado un desafío complejo pero muy atractivo, permitiendo aprender, estudiar y comparar una arquitectura tradicional con una más moderna.

Se ha profundizado en la gestión de infraestructura, abarcando tanto componentes de computación como de red, ampliamente reconocidos en el sector. La construcción de estos elementos como código resultó sumamente gratificante, ya que permitió una intervención manual nula, creando un entorno inmutable y reproducible con un coste basado en el uso real. La posibilidad de levantar un entorno de laboratorio desde casa, trabajar en el proyecto y desmantelarlo al finalizar la sesión, pagando un precio lo más ajustado posible, ha sido fundamental para la realización del proyecto.

El empaquetado de código como imágenes de contenedor, estándar en la industria actual, facilita la compartición de bloques de código independientes. Poder utilizar diferentes versiones de una misma aplicación o fragmento de código agiliza significativamente las pruebas finales, en las que se compararon diversas tecnologías de comunicación. El uso de herramientas de orquestación para estos contenedores se vuelve esencial al manejar numerosas funciones. En este aspecto, Kubernetes se consolidó como una herramienta por defecto para la gestión.

Al añadir una capa adicional con Knative, el escalado de funciones se simplifica enormemente, delegando el despliegue de nuevas instancias o el balanceo de carga entre revisiones en este software de FaaS, todo de manera automatizada. Aunque inicialmente es más complejo que un despliegue monolítico, la capacidad de ajustar recursos dinámicamente y gestionar fallos del sistema justifica completamente su uso.

Cabe destacar que, en las fases iniciales del proyecto, se seleccionó Apache OpenWhisk como la herramienta para implementar una arquitectura basada en microservicios debido a su popularidad por aquel entonces. Sin embargo, a lo largo del desarrollo del proyecto, en fases más avanzadas, surgieron numerosas dificultades debido a la falta de documentación y soporte. Esto llevó incluso a tener que abrir una incidencia en el repositorio de GitHub del proyecto para solicitar ayuda con una operación básica, el despliegue de sus componentes [56]. Anecdóticamente, otro usuario contactó al autor por correo electrónico meses después, preguntando cómo se había logrado el despliegue final en un clúster de Kubernetes con EKS.

Como resultado, se tomó la decisión de detener la implementación de la arquitectura con esta herramienta y usar Knative. Se aporta información sobre el trabajo realizado con Apache OpenWhisk en el anexo.

Una de las conclusiones más evidentes de este trabajo es la notable mejora en flexibilidad que ofrecen los microservicios en comparación con una arquitectura monolítica. En la arquitectura original en PHP, agregar nuevas funcionalidades o escalar ciertas partes del sistema conlleva modificaciones significativas en toda la base de código. Con la nueva arquitectura propuesta, el desacoplamiento de componentes permite que los diferentes módulos se desarrollen,

despliegen y escalen de manera independiente, reduciendo el tiempo de desarrollo y pruebas, y minimizando el riesgo de introducir errores que afecten al sistema global. Sin embargo, esto conlleva una mayor complejidad y desafíos adicionales, como la gestión de peticiones y posibles problemas de comunicación de red durante la ejecución del programa.

Relacionado con este punto está el impacto en el rendimiento de la comunicación entre microservicios. Durante el proyecto, se probaron diferentes protocolos y tecnologías de mensajería, como HTTP, AMQP, y el uso de herramientas de caché en RAM para la comunicación. Se observó que a medida que la arquitectura crece en tamaño y complejidad, aumenta la propensión a errores no solo relacionados con la aplicación y su lógica, sino también con componentes externos. Lo mismo ocurre con el rendimiento: es de esperar que, a mayor número de comunicaciones entre servicios, mayor sea el tiempo añadido al procesamiento de una simulación, aunque esto puede mitigarse y compensarse.

El uso de Julia se destaca especialmente en componentes que requieren cálculos intensivos, gracias a su rendimiento comparable al de lenguajes de bajo nivel y su facilidad para manejar operaciones matriciales. No obstante, en este proyecto no se hizo un uso intensivo de esta capacidad particular. Python, por otro lado, aportó una rica variedad de bibliotecas y una extensa comunidad, permitiendo un desarrollo acelerado de funciones que operan con herramientas como RabbitMQ y PostgreSQL.

6.1 Impacto del proyecto

Durante el desarrollo del proyecto, se ha podido profundizar y entender que la modificación de la arquitectura desde la que se ha partido, y el hecho de poder colaborar en una herramienta con un marcado componente social y tan relacionado con la industria y la ecología, conlleva una serie de implicaciones y aportaciones de índole social, ambiental y económico:

Implicaciones tecnológicas e industriales

El cambio a una arquitectura de microservicios/serverless mejora notablemente la flexibilidad y escalabilidad del simulador. Esto permite un despliegue más sencillo y eficiente en entornos de nube. Hacer esto puede optimizar los recursos utilizados y mejorar el rendimiento a gran escala. Desde una perspectiva industrial, esta transición facilita la integración con otras aplicaciones y servicios, promueve la innovación y la evolución del software a nuevas necesidades del sector energético.

Implicaciones económicas

La arquitectura propuesta en el proyecto permite un modelo de costos que se ajusta al uso, lo cual puede resultar en ahorros significativos para los grupos encargados del simulador. Esto puede resultar particularmente relevante para la sustentabilidad económica de proyectos en el ámbito de las energías renovables, donde la optimización de recursos y costos puede determinar la viabilidad de proyectos a largo plazo.

Implicaciones ambientales

La hipotética eficiencia mejorada del simulador fotovoltaico, al ser más exacto en determinados casos, más rápido en la simulación de diversos escenarios, y más modular, puede contribuir a un mejor diseño y operación de plantas solares. Esto se traduce en una mejor integración de la energía solar en la matriz energética, reduciendo la dependencia de fuentes no renovables y, por ende, disminuyendo la huella de carbono.

Contribución a los Objetivos de Desarrollo Sostenible (ODS)

El trabajo de fin de grado contribuye indirectamente a los Objetivos de Desarrollo Sostenible aprobados en la Agenda 2030 por la ONU.

- **ODS 7: Energía Asequible y No Contaminante:** Al mejorar la eficiencia y accesibilidad del simulador, este proyecto apoya el desarrollo y uso de energías limpias, contribuyendo al objetivo de garantizar el acceso de todas las personas a una energía asequible, segura, sostenible y moderna.

- **ODS 9: Industria, Innovación e Infraestructura:** El modelo de microservicios serverless fomenta la innovación continua y el desarrollo de infraestructuras resilientes y sostenibles.

- **ODS 11: Ciudades y Comunidades Sostenibles:** La modificación y mejora del simulador contribuye y apoya al uso de este para estudiar la viabilidad y transitar a un modelo energético basado en energía fotovoltaica también en las ciudades, lo que se traduce en comunidades más sostenibles, limpias y asequibles.

- **ODS 13: Acción por el Clima:** Al contribuir a una mejor planificación y operación de instalaciones fotovoltaicas, el proyecto apoya medidas para combatir el cambio climático y sus efectos.

El compromiso con tecnologías de código abierto en este proyecto también alinea con los principios de responsabilidad social y tecnológica, promoviendo el acceso libre al conocimiento y la colaboración global.

6.2 Trabajo futuro

Este trabajo de fin de grado tiene como objetivo servir de referencia y brindar lecciones aprendidas para futuras adaptaciones y no solo, en la propia evaluación de la migración de SISIFO desde su microservicio monolítico en MATLAB a un ecosistema serverless en Julia. A lo largo del proyecto se han identificado áreas de mejora y aspectos a explorar que, debido a limitaciones de alcance y tiempo, no se han implementado completamente, aunque se han probado en mayor o menor medida.

Precompilación de funciones Julia

Uno de los problemas conocidos en el ecosistema de Julia es el tiempo de espera experimentado en el "time to first plot" o tiempo de ejecución inicial debido al precompilado. Cuando se ejecuta una función por primera vez, Julia necesita compilar el código Just-In-Time (JIT), lo que puede provocar demoras perceptibles en programas que deben arrancar rápidamente o en scripts que se ejecutan una sola vez. Aunque el precompilado mejora el rendimiento optimizando el código durante la ejecución, este tiempo inicial puede ser problemático en casos específicos, como aplicaciones interactivas o scripts que requieren respuesta inmediata.

La comunidad está trabajando constantemente para mejorar este aspecto, desarrollando técnicas y herramientas que minimicen estas latencias y mejoren la experiencia del usuario final. Aunque existen formas de mitigar estos tiempos de espera, en el contexto de la arquitectura con Knative, donde es posible desplegar funciones bajo demanda sin necesidad de tenerlas previamente desplegadas, las funciones escritas en Julia enfrentan un desafío significativo.

Ejecución asíncrona de funciones

La comunicación asíncrona entre microservicios ofrece varias ventajas clave en comparación con los métodos síncronos [57]. Uno de los principales beneficios es la reducción de la dependencia temporal entre servicios, lo que permite que un microservicio envíe un mensaje y continúe procesando sin esperar una respuesta inmediata. Esto mejora la resiliencia del sistema, ya que los servicios no se bloquean si hay fallos o latencias en otros servicios.

Además, la comunicación asíncrona facilita una mayor escalabilidad, permitiendo que los mensajes sean enrutados, escalonados y procesados por múltiples instancias de un servicio receptor según sea necesario. Este enfoque también permite implementar patrones de diseño avanzados, como la gestión de colas de mensajes y el procesamiento de eventos, lo que puede resultar en sistemas más eficientes en el manejo de grandes volúmenes de datos o solicitudes, como en el caso del simulador.

El principal desafío aquí es la gran cantidad de funciones de SISIFO. El resultado final de una simulación es el conjunto de llamadas a estas funciones. Por lo tanto, al implementar un funcionamiento y operación totalmente asíncrona, es esencial identificar de manera efectiva los diferentes outputs de las funciones para componer el resultado final de forma consistente, sin mezclar distintas llamadas.

Uso de Knative Eventing

Eventing es el segundo componente esencial de Knative. Ofrece importantes ventajas al facilitar la construcción de aplicaciones impulsadas por eventos. Proporciona una capa de abstracción que permite a las aplicaciones responder automáticamente a eventos internos y externos de manera escalable, haciendo uso del estándar *CloudEvents* [58]. Esto facilitaría enormemente la implementación de una arquitectura asíncrona partiendo de lo ya desarrollado en el proyecto.

Protocolos binarios como gRPC

Una posible alternativa a considerar, en seguimiento del uso de Memcached para mitigar el impacto de la serialización y las peticiones HTTP con JSON, es gRPC (Google Remote Procedure Calls). gRPC es una implementación de llamadas a procedimiento remoto que utiliza HTTP/2, ofreciendo características como multiplexación, compresión de cabeceras y priorización de solicitudes, mejorando así la velocidad y eficiencia en la comunicación.

Esta tecnología, emplea un formato de serialización binaria, *Protocol Buffers*, que reduce el uso de ancho de banda y mejora el rendimiento respecto a formatos más verbosos como JSON. También facilita el desarrollo de APIs fuertemente tipadas, reduciendo errores y aumentando la coherencia en la comunicación entre servicios.

Despliegue en infraestructura de la Escuela o del Grupo de Investigación en Sistemas Fotovoltaicos

El proyecto actualmente se ha desplegado en AWS. Sin embargo, por motivos regulatorios, académicos o para reducir costes y mejorar la disponibilidad, podría ser beneficioso desplegar los componentes de infraestructura en un entorno controlado y propiedad de la escuela. El código Terraform utilizado en este proyecto está configurado para implementar recursos en AWS, pero existen muchos otros providers para diferentes clouds y para entornos on-premises como OpenStack, Oracle Private Cloud, VMware vSphere, y Proxmox, entre otros.

Despliegue apto para producción

A lo largo del proyecto se ha resaltado que los despliegues no son aptos para ambientes críticos con alta carga de trabajo. Al ofrecer un servicio a los usuarios, es crucial garantizar alta disponibilidad y recuperación ante desastres. Acciones como desplegar Memcached en todos los nodos del clúster de Kubernetes como un DaemonSet o desplegar RabbitMQ con tolerancia a fallos son esenciales para aumentar la resiliencia del simulador.

La seguridad es igualmente importante, por lo que todos los componentes deben implementarse con autenticación y autorización, siguiendo el principio de Zero Trust para mitigar riesgos de agentes maliciosos externos. Esto incluye autenticación en RabbitMQ o Memcached, encriptación de tráfico con TLS entre microservicios, y políticas de red estrictas.

Además, si la arquitectura se publica en Internet, se deben controlar las conexiones entrantes con TLS (HTTPS), y posiblemente registrar dominios o subdominios con sus respectivos certificados.

Observabilidad de la plataforma

Para analizar el rendimiento de la arquitectura y evitar el sobredimensionamiento o la infrautilización, es fundamental implementar un sistema de observabilidad robusto en la plataforma. Esto permite recopilar, analizar y visualizar datos sobre el estado y comportamiento de los microservicios y la infraestructura subyacente. Utilizando herramientas de monitoreo y logging, como Prometheus para la recolección de métricas, Grafana para la visualización de datos en tiempo real, y OpenSearch para los logs, es posible obtener una visión detallada del funcionamiento interno del sistema.

Estas herramientas ayudan a identificar cuellos de botella, errores o anomalías en el rendimiento, facilitando la toma de decisiones sobre la asignación de recursos. Además, la capacidad de establecer alertas asegura que los equipos encargados de la operación y mantenimiento puedan responder rápidamente a incidencias antes de que afecten a los usuarios finales.

Robustecimiento del código

Este proceso busca hacer que el software sea más resistente a errores y fallos, mejorando su capacidad para manejar situaciones inesperadas o extremos sin colapsar. Aplicado al código de este proyecto, incluye prácticas como la validación y limpieza de entradas, el manejo adecuado de excepciones, pruebas exhaustivas bajo diversas condiciones, y la implementación de mecanismos de recuperación ante errores. El código generado en este proyecto carece de máxima robustez, debido en parte a la inexperiencia del autor con Julia.

Aspectos como el tipado de variables para mejorar no solo el rendimiento, sino también la legibilidad, la implementación de pruebas unitarias, y la separación del código en paquetes más reutilizables y atómicos siguiendo las convenciones de Julia, son ejemplos de trabajo a desarrollar

Anexo

A.1 Apache OpenWhisk

Apache OpenWhisk es una plataforma abierta y distribuida para ejecutar funciones en respuesta a eventos [23]. Está diseñada para el modelo de computación sin servidor (serverless computing), lo que permite a los desarrolladores implementar código que se ejecuta únicamente en reacción a eventos o solicitudes, sin la necesidad de administrar servidores o infraestructura subyacente (Figura 73).

Como se indica en el apartado de conclusiones, en las fases iniciales del proyecto, se seleccionó Apache OpenWhisk como la herramienta para implementar una arquitectura basada en microservicios. Sin embargo, la falta de documentación y soporte y la dificultad para operar con la herramienta en un entorno de Kubernetes hicieron que se tomase la decisión de detener la implementación de la arquitectura con esta opción.

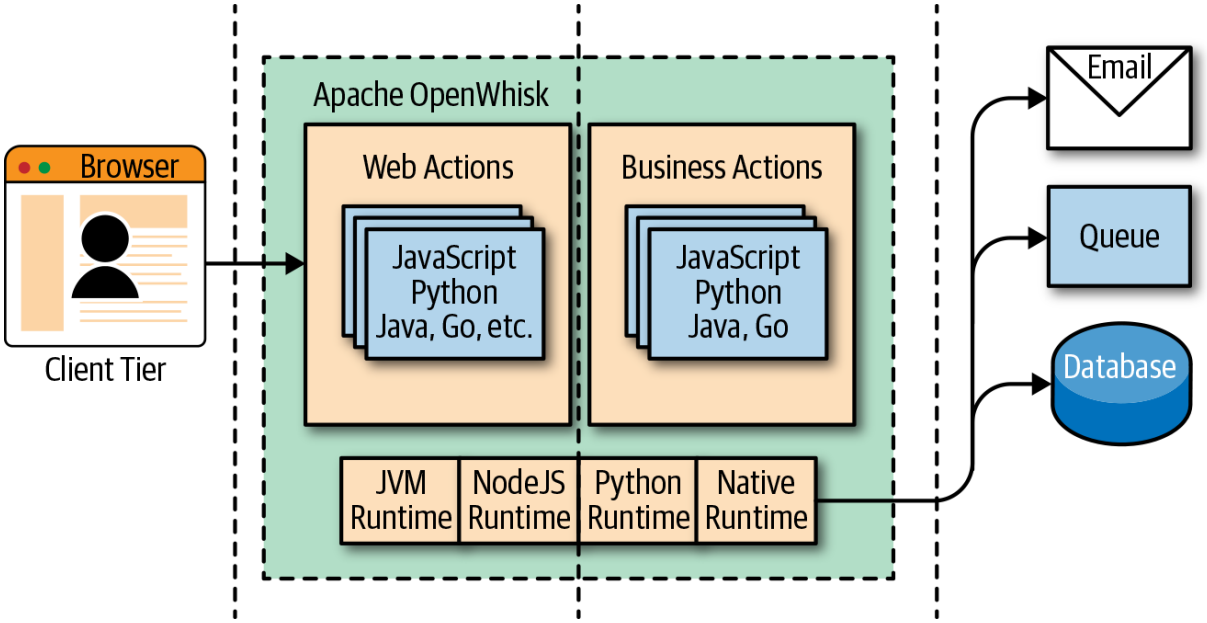


Figura 73. Arquitectura de Apache OpenWhisk *(Fuente: Abhishek Ghosh)*

A.1.1. Comparativa entre Apache OpenWhisk y Knative

Considerando los diversos aspectos recopilados durante la etapa de documentación, y antes de decidir descontinuar el uso de OpenWhisk en el proyecto, se presentan los puntos clave que se tuvieron en cuenta para comparar ambos productos. Se enumeran únicamente los factores diferenciadores:

Apache OpenWhisk:

- Puede ser desplegado en diversas plataformas, como Kubernetes, Mesos, máquinas virtuales, o incluso servidores físicos.
- Es un sistema independiente que no depende necesariamente de Kubernetes para funcionar.
- Está optimizado para ejecutar funciones pequeñas y rápidas en respuesta a eventos.
- Las funciones son sin estado y están diseñadas para ser efímeras.
- Utiliza un sistema de activadores y reglas donde los eventos disparan la ejecución de acciones (funciones).
- Escala horizontalmente para manejar un gran número de invocaciones simultáneas.
- Cuenta con un sistema modular que permite la integración con otros sistemas mediante paquetes y extensiones.
- Como proyecto de la Apache Software Foundation, tiene una comunidad dedicada a su desarrollo y mejora continua.
- Ofrece soporte a través de foros, listas de correo y documentación en línea.

Knative:

- Está diseñado para ejecutarse exclusivamente en Kubernetes y requiere un clúster subyacente de Kubernetes para su operación.
- Su instalación implica la adición de componentes a un clúster existente de Kubernetes.
- Está compuesto por dos componentes principales: *serving* y *eventing*, que flexibilizan su uso.
- Permite la gestión tanto de funciones individuales como de aplicaciones completas, utilizando contenedores, lo que proporciona más flexibilidad respecto al tipo de cargas de trabajo que pueden ser gestionadas.
- Soporta autoescalado basado en la demanda y puede escalar de cero a N, ajustando automáticamente el número de réplicas según el tráfico.
- Se beneficia del ecosistema de Kubernetes, integrándose fácilmente con herramientas y servicios como Prometheus, Grafana e Istio.
- Está respaldado por Google y otras organizaciones líderes, lo que garantiza un soporte sólido y rápido desarrollo de nuevas funcionalidades.
- Posee una comunidad activa dentro del ecosistema de Kubernetes, con contribuciones de múltiples empresas y desarrolladores.

Desde una perspectiva personal del autor, aunque OpenWhisk puede funcionar con Kubernetes, su despliegue en un clúster no es una tarea trivial y la documentación es relativamente escasa para este tipo de configuración.

Knative, por otro lado, está diseñado para aprovechar las capacidades de Kubernetes, utilizando sus objetos y API. Como se describió anteriormente, no es complicado utilizar sus componentes.

Ambos proyectos son de código abierto y su desarrollo se lleva a cabo de forma transparente en GitHub. Sin embargo, a la fecha de redacción de este documento, Knative muestra mucha

más actividad. Por ejemplo, solo el componente de "serving" de Knative tiene más *commits* que el repositorio principal de OpenWhisk. Aunque esto no garantiza la calidad de un proyecto, podría indicar su salud y continuidad.

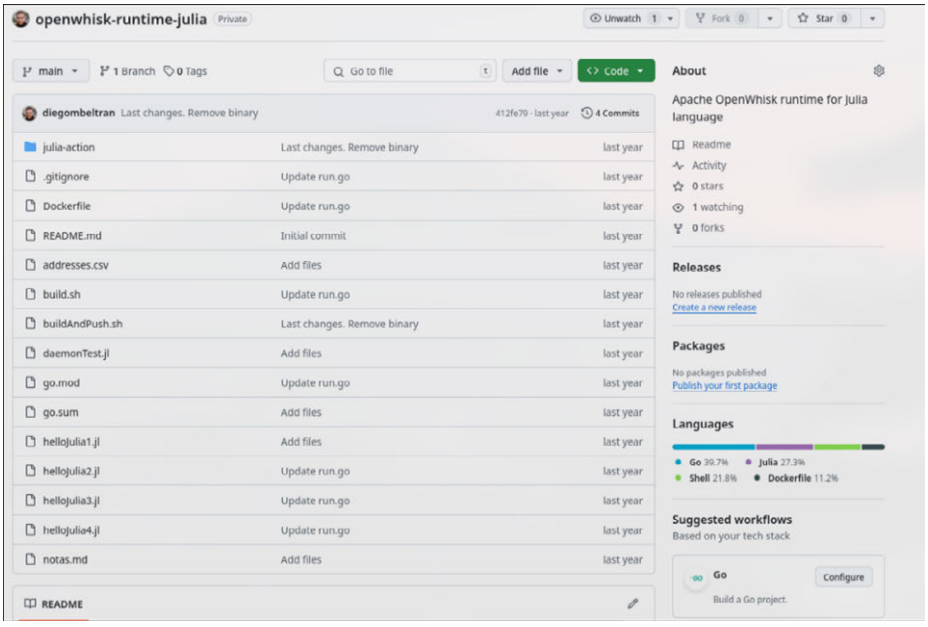
Por lo general, el desarrollo de Knative es más constante, cuenta con mayor soporte y más contribuidores. A modo ilustrativo, OpenWhisk ha publicado dos versiones estables con cuatro años de diferencia entre ellas, mientras que en Knative el intervalo entre sus dos últimas versiones estables es de menos de un mes. El lanzamiento de versiones es mucho más continuo en el tiempo.

La documentación de Knative es más extensa, detallada y precisa. Incluye numerosos ejemplos y dispone de un libro muy detallado publicado por O'Reilly, el cual ha sido utilizado frecuentemente como fuente bibliográfica de referencia a lo largo del proyecto.

A.1.2. Trabajo realizado con OpenWhisk

Se han desarrollado dos repositorios relacionados con OpenWhisk. El primero contiene la infraestructura necesaria para desplegar OpenWhisk utilizando Terraform. Este repositorio también ha sido adaptado para su uso con Knative. La principal diferencia radica en la segmentación de los nodos del clúster en grupos diferenciados mediante "taints" y "tolerations", distinguiendo entre nodos "workers" y nodos "invokers".

En OpenWhisk, los "invokers" son responsables de gestionar la ejecución de las funciones (conocidas como "acciones"), mientras que los "workers" son nodos que llevan a cabo tareas específicas.



Captura 74. Repositorio de runtime Openwhisk *(Fuente: Elaboración propia)*

El segundo repositorio (Captura 74) alberga el código para implementar un *runtime* de OpenWhisk para el lenguaje Julia.

Un runtime de OpenWhisk es un entorno de ejecución predefinido, basado en contenedores, que proporciona las dependencias y configuraciones necesarias para ejecutar funciones en un lenguaje de programación específico, en este caso, Julia.

Esto facilita la ejecución de funciones sin que el desarrollador deba preocuparse por la gestión del entorno subyacente.

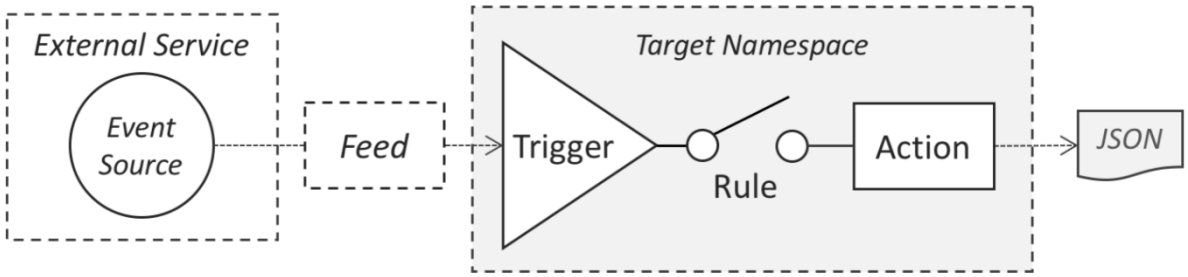


Figura 75. OpenWhisk Programming Model *(Fuente: Apache OpenWhisk Project)*

Aunque el runtime está diseñado para ejecutar funciones en Julia, está escrito en el lenguaje de programación Go. Esto es debido a que los requisitos de implementación de un runtime, según la documentación y ejemplos de OpenWhisk, son fácilmente replicables en Go. Los pasos para seguir son los siguientes:

- Crear un servidor HTTP que escuche las invocaciones de OpenWhisk.
- Implementar un mecanismo para ejecutar scripts de Julia dentro del contenedor.
- Convertir las solicitudes y respuestas JSON entre OpenWhisk y Julia.
- Construir una imagen Docker que contenga Go y Julia.
- Configurar OpenWhisk para utilizar la imagen Docker personalizada como runtime.

El resultado final de este proceso es la capacidad de definir y ejecutar fragmentos de código a través del runtime previamente creado, configurado por defecto en el clúster. Este enfoque difiere del de Knative, donde los servicios son autónomos y no dependen de un runtime externo (Figura 75).

7 Referencias

[1] J. S.-S. V. S. C.-C. T. A. K. Q. P. V. S. J. C. K. K. N. Y. J. E. G. R. A. P. I. S. D. A. P. Eric Jonas, «Cloud Programming Simplified: A Berkeley View on Serverless Computing,» 2019.

[2] Amazon, «What is Serverless Computing?,» 2024. [En línea]. Available: https://aws.amazon.com/es/what-is/serverless-computing/. [Último acceso: 10 2023].

[3] Global Market Insights, «Arquitectura sin servidores,» 2022. [En línea]. Available: https://www.gminsights.com/es/industry-analysis/serverless-architecture-market.

[4] Amazon, «Serverless Architectures with AWS Lambda,» 2024. [En línea]. Available: https://docs.aws.amazon.com/pdfs/whitepapers/latest/serverless-architectures-lambda/serverless-architectures-lambda.pdf.

[5] «Serverless Open Source Frameworks,» 13 04 2020. [En línea]. Available: https://www.cncf.io/blog/2020/04/13/serverless-open-source-frameworks-openfaas-knative-more/. [Último acceso: 10 2023].

[6] UPM, «El programa Horizonte 2020 destaca en su web el proyecto PVCROPS,» 2020. [En línea]. Available: https://www.upm.es/UPM/SalaPrensa/Noticias_de_investigacion?id=45e8a7fa895f3510VgnVCM10000009c7648a____&fmt=detail&prefmt=articulo.

[7] MASLOWATEN, «Presentación General MASLOWATEN,» 2020. [En línea]. Available: https://www.maslowaten.eu/files/presentaciones/Chile/Maslowaten_Chile.pdf.

[8] Mathworks, «Comparison of MATLAB and Other OO Languages,» [En línea]. Available: https://es.mathworks.com/help/matlab/matlab_oop/matlab-vs-other-oo-languages.html. [Último acceso: 10 2023].

[9] Mathworks, «MATLAB Pricing Model,» [En línea]. Available: https://es.mathworks.com/pricing-licensing.html. [Último acceso: 11 2024].

[10] Amazon, «AWS Lambda Release Notes,» 13 11 2014. [En línea]. Available: https://aws.amazon.com/es/releasenotes/release-aws-lambda-on-2014-11-13/. [Último acceso: 10 2023].

[11] Microsoft, «Announcing general availability of Azure Functions,» 15 11 2016. [En línea]. [Último acceso: 10 2023].

[12] Google, «Cloud Functions Release Notes,» 09 03 2017. [En línea]. Available: https://cloud.google.com/functions/docs/release-notes#March_09_2017. [Último acceso: 10 2023].

[13] R. Osnat, «The History of Kubernetes,» 25 07 2024. [En línea]. Available: https://www.aquasec.com/blog/kubernetes-history-how-it-conquered-cloud-native-orchestration/. [Último acceso: 11 2024].

[14] Deloitte, «¿Qué es Serverless?,» 11 02 2022. [En línea]. Available: https://www.deloitte.com/es/es/services/consulting/blogs/todo-tecnologia/que-es-serverless-cloud.html. [Último acceso: 10 2023].

[15] S. Bustos, «5 casos de éxito de empresas que usan AWS Lambda,» [En línea]. Available: https://codster.io/blog/aws-lambda/5-historias-de-exito-de-empresas-de-aws-lambda/. [Último acceso: 12 2024].

[16] C. Harris, «Comparación entre la arquitectura monolítica y la arquitectura de microservicios,» [En línea]. Available: https://www.atlassian.com/es/microservices/microservices-architecture/microservices-vs-monolith. [Último acceso: 10 2023].

[17] A. Kaplunovich, «ToLambda - Automatic Path to Serverless Architectures,» 05 2019. [En línea]. Available: https://www.researchgate.net/publication/335945256_ToLambda--Automatic_Path_to_Serverless_Architectures. [Último acceso: 10 2023].

[18] Amazon, «AWS Lambda Documentation,» [En línea]. Available: https://docs.aws.amazon.com/lambda/. [Último acceso: 12 2024].

[19] L. Minvielle, «Top 9 Serverless Function (FaaS) Providers,» 27 08 2024. [En línea]. Available: https://genezio.com/blog/best-serverless-faas-providers/. [Último acceso: 12 2024].

[20] P. F. M. Soto, «Serverless for private clouds (or managing the server for a Serverless (!?) app),» 30 4 2017. [En línea]. Available: https://medium.com/@pfernandom/serverless-for-private-clouds-or-managing-the-server-for-a-serverless-app-f9321e45a910https://medium.com/@pfernandom/serverless-for-private-clouds-or-managing-the-server-for-a-serverless-app-f9321e45a910. [Último acceso: 10 2023].

[21] OpenFaaS Project, «OpenFaaS Documentation,» [En línea]. Available: https://docs.openfaas.com/. [Último acceso: 12 2024].

[22] Redbee, «FaaS: Simplificando el desarrollo de aplicaciones sin servidor,» 18 08 2023. [En línea]. Available: https://medium.com/redbee/faas-simplificando-el-desarrollo-de-aplicaciones-sin-servidor-87a201d3f94a. [Último acceso: 11 2023].

[23] Apache OpenWhisk, «Apache OpenWhisk,» [En línea]. Available: https://openwhisk.apache.org/. [Último acceso: 12 2022].

[24] GitHub, «Apache OpenWhisk Releases,» [En línea]. Available: https://github.com/apache/openwhisk/tags. [Último acceso: 10 2024].

[25] Apache, «OpenWhisk Documentation,» [En línea]. Available: https://openwhisk.apache.org/documentation.html. [Último acceso: 10 2023].

[26] Google, «Knative Serving en consola de Google Cloud,» [En línea]. Available: https://cloud.google.com/kubernetes-engine/enterprise/knative-serving/docs/console?hl=es-419. [Último acceso: 10 2024].

[27] Knative Project, «Knative Case Studies,» 2024. [En línea]. Available: https://knative.dev/docs/about/case-studies/. [Último acceso: 12 2024].

[28] Docker Project, «Docker Documentation,» 2024. [En línea]. Available: https://docs.docker.com/get-started/docker-overview/. [Último acceso: 10 2023].

[29] IBM, «The history of Kubernetes,» 2 11 2023. [En línea]. Available: https://www.ibm.com/think/topics/kubernetes-history. [Último acceso: 12 2024].

[30] Kubernetes Project, «Kubernetes Documentation,» [En línea]. Available: https://kubernetes.io/docs/home/. [Último acceso: 10 2023].

[31] T. Haigh, «Cleve Moler: Mathematical Software Pioneer and Creator of Matlab,» 2008. [En línea]. Available: https://www.tomandmaria.com/Tom/Writing/MolerBio.pdf. [Último acceso: 12 2024].

[32] Mathworks, «Getting Started with MATLAB,» [En línea]. Available: https://es.mathworks.com/help/matlab/getting-started-with-matlab.html. [Último acceso: 12 2024].

[33] javatpoint.com, «Advantages and Disadvantages of MATLAB Programming Language,» [En línea]. Available: https://www.javatpoint.com/advantages-and-disadvantages-of-matlab. [Último acceso: 12 2024].

[34] S. K. V. B. S. A. E. Jeff Bezanson, «Why We Created Julia,» 14 2 2012. [En línea]. Available: https://julialang.org/blog/2012/02/why-we-created-julia/. [Último acceso: 10 2023].

[35] Julia Project, «Julia 1.9 Documentation,» [En línea]. Available: https://docs.julialang.org/en/v1.9/. [Último acceso: 12 2024].

[36] Z. Keita, «Introducing Julia with a comparison to Python and R,» 15 12 2022. [En línea]. Available: https://dagshub.com/blog/introduction-to-julia-and-comparison-to-python-and-r/. [Último acceso: 12 2024].

[37] A. S. Alberca, «Pácticas de Análisis Matemático con Julia,» [En línea]. Available: https://aprendeconalf.es/analisis-practicas-julia/practicas-analisis-julia.pdf. [Último acceso: 11 2024].

[38] Julia Project, «JuliaHub,» [En línea]. Available: https://juliahub.com/. [Último acceso: 10 2023].

[39] I. Olusheye, «Interoperability in Julia,» 20 05 2022. [En línea]. Available: https://forem.julialang.org/ifihan/interoperability-in-julia-1m26. [Último acceso: 12 2024].

[40] S. Karpinski, «Some Julia Growth and Usage,» 4 4 2024. [En línea]. Available: https://discourse.julialang.org/t/some-julia-growth-usage-stats/112547. [Último acceso: 12 2024].

[41] OpenEDG | Open Education and Development Group, LLC, «https://pythoninstitute.org/about-python,» [En línea]. Available: OpenEDG | Open Education and Development Group, LLC . [Último acceso: 01 12 2024].

[42] T. Peters, «The Zen of Python,» 06 1999. [En línea]. Available: http://www.thezenofpython.com/. [Último acceso: 10 2023].

[43] TIOBE, «The Python Programming Language,» 12 2024. [En línea]. Available: https://www.tiobe.com/tiobe-index/python/. [Último acceso: 12 2024].

[44] PyPi Project, «PyPi.org,» 12 2024. [En línea]. Available: https://pypi.org/. [Último acceso: 12 2024].

[45] NumPy Project, «NumPy,» [En línea]. Available: https://numpy.org/. [Último acceso: 12 2024].

[46] SciPy Project, «SciPy,» [En línea]. Available: https://scipy.org/. [Último acceso: 12 2024].

[47] TensorFlow, «TensorFlow,» [En línea]. Available: https://www.tensorflow.org/?hl=es-419. [Último acceso: 12 2024].

[48] Universidad Europea, «Usos de Python: ¿qué ofrece este lenguaje de programación?,» 5 07 2023. [En línea]. Available: https://universidadeuropea.com/blog/usos-python/. [Último acceso: 10 2023].

[49] IBM, «¿Qué es Terraform?,» [En línea]. Available: https://www.ibm.com/es-es/topics/terraform. [Último acceso: 15 2024].

[50] HashiCorp, «Terraform Documentation,» [En línea]. Available: https://developer.hashicorp.com/terraform/docs. [Último acceso: 10 2023].

[51] RabbitMQ Project, «RabbitMQ Documentation,» [En línea]. Available: https://www.rabbitmq.com/docs. [Último acceso: 12 2024].

[52] Kinsta, «Memcached vs Redis,» [En línea]. Available: https://kinsta.com/blog/memcached-vs-redis/. [Último acceso: 12 2024].

[53] Amazon, «EKS Documentation,» [En línea]. Available: https://docs.aws.amazon.com/eks/latest/userguide/getting-started.html. [Último acceso: 10 2023].

[54] Knative Project, «Knative Serving Documentation,» [En línea]. Available: https://knative.dev/docs/serving/. [Último acceso: 10 2024].

[55] Amazon, «¿Qué es .NET?,» [En línea]. Available: https://aws.amazon.com/es/what-is/net/. [Último acceso: 16 2024].

[56] D. M. Beltrán, «Can't deploy on EKS,» 17 06 2023. [En línea]. Available: https://github.com/apache/openwhisk-deploy-kube/issues/766.

[57] D. Bevans, «Asynchronous vs. Synchronous Programming: Key Similarities and Differences,» 24 10 2024. [En línea]. Available: https://www.mendix.com/blog/asynchronous-vs-synchronous-programming/. [Último acceso: 10 2024].

[58] Cloudevents Project, «Cloudevents,» [En línea]. Available: https://cloudevents.io/. [Último acceso: 12 2024].

[59] C. C. D. M. Federico M Giorgi 1, «The R Language: An Engine for Bioinformatics and Data Science,» 2022.

[60] R Project, «What is R?,» [En línea]. Available: https://www.r-project.org/about.html. [Último acceso: 10 2023].

[61] tmtoolkit Project, «tmtoolkit Docs,» [En línea]. Available: https://tmtoolkit.readthedocs.io/en/latest/rinterop.html. [Último acceso: 12 2024].

[62] ggplot Project, «ggplot Docs,» [En línea]. Available: https://ggplot2.tidyverse.org/. [Último acceso: 12 2024].

[63] gRPC Project, «Introduction to gRPC,» [En línea]. Available: https://grpc.io/docs/what-is-grpc/introduction/. [Último acceso: 10 2024].