Universidad Politécnica
de Madrid

Escuela Técnica Superior de
Ingenieros Informáticos

Grado en Ingeniería Informática

Trabajo Fin de Grado

Diseño y Despliegue de una Solución CI/CD y Serverless con AWS

Autor: Santiago Dario Montero Cabezas

Tutor(a): Sonia Frutos Cid

Madrid, mayo 2024

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

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Diseño y Despliegue de una Solución CI/CD y Serverless con AWS
Mayo 2024

Autor: Santiago Dario Montero Cabezas

Tutor:
Sonia Frutos Cid
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En el trabajo presentado a continuación se realiza el desacoplamiento de una aplicación monolítica hacia una arquitectura serverless con una solución CI/CD integrada.

Para la realización de este proyecto se usan los servicios proveídos por Amazon Web Services, mediante los cuales se construirá el proyecto entero. Se realiza una selección de servicios a usar, el diseño de una arquitectura y la estimación de costo para la arquitectura diseñada.

Entre los servicios usados para la arquitectura se encuentran AWS Lambda, API Gateway, AWS CloudFront, Amazon S3 y DynamoDB. La automatización de la creación de la infraestructura se realiza usando una plantilla de AWS SAM.

Una vez definida la plantilla de la arquitectura se construye un pipeline para automatizar el despliegue de la infraestructura cuando se produzcan cambios en un repositorio. Este también se realiza a través de una plantilla de AWS SAM. Los servicios usados para la construcción del pipeline CI/CD se encuentran AWS CodePipeline, AWS CodeCommit y AWS CodeBuild.

Abstract

In the presented work, the decoupling of a monolithic application into a serverless architecture with an integrated CI/CD solution is carried out.

For the realization of this project, the services provided by Amazon Web Services are used, through which the entire project will be built. A selection of services to be used is made, the design of an architecture is created, and the cost estimation for the designed architecture is performed.

Among the services used for the architecture are AWS Lambda, API Gateway, AWS CloudFront, Amazon S3, and DynamoDB. The automation of the infrastructure creation is done using an AWS SAM template.

Once the architecture template is defined, a pipeline is built to automate the deployment of the infrastructure when changes are made in a repository. This is also done through an AWS SAM template. The services used for building the CI/CD pipeline include AWS CodePipeline, AWS CodeCommit, and AWS CodeBuild.

Tabla de contenidos

1 Introducción

1.1 Contexto y Motivación del Proyecto

En los últimos años, debido a la transformación digital, se ha aumentado el uso y la adopción de tecnologías cloud alrededor del mundo debido a sus beneficios. Entre sus beneficios podemos encontrar la escalabilidad, flexibilidad y reducción de costes operativos.

A pesar de esto, podemos encontrar arquitecturas tradicionales basadas en aplicaciones monolíticas que presentan serias limitaciones ya que suelen ser rígidas, costosas de mantener y presentan también problemas a la hora de escalar frente a picos de alta demanda por parte de los usuarios. Así, las arquitecturas serverless surgen como una solución revolucionaria debido a su escalabilidad automática, menor carga de mantenimiento y aún más optimización de costes operativos. Las arquitecturas serverless se basan en la implementación de funciones que se ejecutan en respuesta a eventos, como solicitudes HTTP, y solo pagas por el tiempo de ejecución y los recursos usados, como la memoria y el almacenamiento.

Además, es importante mencionar la importancia de prácticas DevOps hoy en día. La implementación de pipelines de integración y desarrollo continuo (CI/CD) facilitan la automatización de procesos de desarrollo y despliegue de software. Amazon Web Services (AWS), nos ofrece una gran variedad de servicios y herramientas que nos permiten la implementación de soluciones serverless y CI/CD.

Es en este contexto en donde se sitúa este Trabajo de Fin de Grado, que tiene como fin el diseño y despliegue de una solución CI/CD y serverless con AWS. La motivación principal de este proyecto tiene que ver con la necesidad de la transformación de una arquitectura monolítica hacia una arquitectura serverless para superar sus limitaciones, abrir las puertas a futuras innovaciones, y además permitir la automatización de la creación de la infraestructura (IaC) que ayude a los desarrolladores a centrarse en la lógica de negocio.

1.2 Objetivos Generales y Específicos

El objetivo de este trabajo es el diseño y despliegue de una arquitectura serverless en AWS, transformando una aplicación web monolítica desarrollada en Node.js conectada a una base de datos relacional en una arquitectura serverless.

Para conseguir este objetivo habrá que:

- Estudiar los servicios cloud AWS de CI/CD.
- Desacoplar una aplicación monolítica en funciones escalables.
- Migrar la base de datos a un servicio serverless.
- Definir un pipeline CI/CD.
- Automatizar de la infraestructura y creación de servicios cloud.
- Estimar costes.

1.3 Estructura del documento

- **Capítulo 1: Introducción –** Presenta el contexto y la motivación del proyecto, los objetivos generales y específicos, y la estructura del documento.

- **Capítulo 2: Estado del arte –** Examina los servicios cloud de AWS para CI/CD, las arquitecturas serverless, el desacoplamiento de las aplicaciones monolíticas y las técnicas de despliegue blue/green.

- **Capítulo 3: Metodología –** Describe el enfoque general del proyecto, las herramientas y tecnologías utilizadas, y la planificación del trabajo.

- **Capítulo 4: Diseño de la Arquitectura -** Analiza la aplicación monolítica existente, diseña la nueva arquitectura serverless, selecciona los servicios AWS adecuados y presenta el esquema de la infraestructura.

- **Capítulo 5: Desarrollo –** Detalla la creación del entorno de desarrollo, el desacoplamiento de la aplicación monolítica, la automatización de la creación de la infraestructura y la definición del pipeline CI/CD.

- **Capítulo 6: Resultados y Conclusiones –** Presenta los resultados y conclusiones, y compara los costes entre la aplicación monolítica y la aplicación serverless.

- **Capítulo 7: Análisis de Impacto -** Analiza el impacto del proyecto a nivel personal, empresarial, social, económico, medioambiental, y su relación con los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030.

2 Estado del arte

En este capítulo se revisarán los conceptos y tecnologías que son importantes para el proyecto.

2.1 Servicios Cloud de AWS para CI/CD

Primero hay que entender que significa CI/CD. Como fue mencionado antes, CI/CD se refiera a la integración y desarrollo continuo cuyo objetivo es mejorar y agilizar el ciclo de vida de desarrollo del software [1]. Ambas forman parte de las practicas DevOps. Están diseñadas para mejorar la calidad del software y acelerar el despliegue a producción, esto también beneficia a disminuir el riesgo de los potenciales errores al momento de desplegar manualmente una aplicación.

La integración continua (CI) es una práctica donde se integra el código continuamente en un repositorio compartido. Esta integración construye y testea el código para asegurar que los cambios que se hagan sean fiables y no causen fallos en el código previo.

Por otro lado, la distribución continua (CD) se refiere a la automatización del despliegue de la aplicación una vez que esta se haya construido y haya pasado las pruebas correspondientes.

Según AWS, el CI/CD puede ser visto como una "tubería" en donde el nuevo código es introducido, por el inicio de la tubería, para pasar por un proceso de construcción, testeo y despliegue, para finalmente entregar código listo para producción [2].

AWS ofrece una gran variedad de herramientas y servicios para la creación de pipelines CI/CD.

2.1.1 AWS CodeCommit

Este es un servicio que te permite alojar repositorios de Git privados, administrados totalmente por AWS y con una alta escalabilidad. [3]

Entre otras funcionalidades, AWS CodeCommit te permite:

- Transferir archivos con HTTPS o SSH hacia o desde el repositorio.
- Almacenar cifrada y redundantemente el repositorio para aumentar la disponibilidad.
- Participar fácilmente en desarrollo de software colaborativo.

Para crear un repositorio se puede usar la consola de Amazon, los SDK o el interfaz de línea de Amazon (AWS CLI).

2.1.2 AWS CodeBuild
Este es un servicio de integración continua que te permite compilar código fuente, testear el código y producir software empaquetado listo para ser desplegado [4]. Es un servicio completamente administrado, lo que quiere decir que no tienes que preocuparte por crear tus servidores de compilación y testeo, ya que AWS te lo proporciona, con la opción de montar entornos de compilación ya creados o proporcionar el tuyo. Además de todo esto, AWS CodeBuild escala automáticamente para que no tengas que preocuparte por colas innecesarias.

2.1.3 AWS CodePipeline
Este servicio es el que te permite definir tu pipeline, es decir, con AWS CodePipeline se define las etapas del proceso de lanzamiento de software [5]. Una vez más, es un servicio totalmente administrado. Lo que nuevamente permite dejar a un lado la creación de nuestros propios servidores.

2.1.4 AWS CodeDeploy
Este servicio administrado, automatiza los despliegues de software en diferentes entornos, como desarrollo, pruebas y producción. Permite el despliegue en distintos servicios propios de AWS, como AWS Lambda, Amazon Elastic Compute Cloud (EC2) o Amazon Elastic Container Service (ECS) [6]. Por último, permite deshacer los despliegues si es necesario.

2.1.5 AWS CloudFormation
Es un servicio con el que se puede implementar y seguir tu infraestructura de AWS controlada y precisamente. Permite el modelado y diseño de tus recursos AWS dentro de tu infraestructura a través de ficheros de texto con formato JSON o YAML [7]. Esto facilita la creación de la infraestructura, y, además, permite replicar fácilmente dicha infraestructura para crear diferentes entornos, como entornos de prueba o producción.

Una de las capacidades más poderosas de CloudFormation es la capacidad de gestionar dependencias entre recursos de manera sofisticada. Puede modelar y gestionar las relaciones y dependencias entre los recursos, de tal manera que los recursos se desplieguen secuencialmente y se mantengan en un estado coherente. Esto beneficia al trabajar con infraestructuras que contienen un número considerable de recursos y servicios. Por último, CloudFormation minimiza la posibilidad de errores y el tiempo de inactividad.

2.2 Arquitecturas Serverless
Las arquitecturas serverless permiten a los desarrolladores centrarse en el código en lugar de en el mantenimiento del servidor. En lugar de gestionar la infraestructura y la lógica del servidor, simplemente se despliega una función que se ejecuta solo cuando se dispara un evento especifico, por ejemplo, una modificación en una base de datos o una solicitud HTTP entrante.

AWS Lambda es el servicio que da vida a esta forma de arquitectura, es capaz de ejecutar código sin tener que preocuparte por la administración de la maquina servidora. Con Lambda, lo único por lo que realmente pagas es el tiempo de ejecución, permitiendo así, optimizar los costos.

Existen también otros servicios que complementan Lambda y que ayudan a construir una arquitectura serverless completa. Entre esos otros servicios, tenemos a ApiGateway que te permite la creación y gestión de APIs, que pueden servir como eventos para lanzar las funciones Lambda. También existe AWS Step Functions, que permite la orquestación de funciones Lambda para crear flujos de trabajo tan grandes y complejos como quieras.

Las arquitecturas serverless tienen varios casos de uso debido a su escalabilidad y flexibilidad. Uno de los casos más comunes es el procesamiento de eventos en tiempo real, donde se utiliza AWS Lambda para el procesamiento en tiempo real de eventos emitidos por servicios como Amazon Kinesis o Amazon DynamoDB Streams [8]. Esto permite reaccionar automáticamente a eventos que pueden considerarse críticos, como transacciones financieras, cambios de inventario o análisis de las redes sociales. Por último, otro caso común es el de aplicaciones web o solo backend de aplicaciones.

2.3 Desacoplamiento de Aplicaciones Monolíticas

2.3.1 Definición de Aplicaciones Monolíticas

Es necesario empezar definiendo que es una aplicación monolítica. Una aplicación monolítica es un tipo de aplicación en el que todos sus componentes funcionan y se despliegan juntos como una sola unidad.

Las aplicaciones monolíticas son caracterizadas por lo siguiente:

- **Módulos fuertemente acoplados:** Todos los módulos y componentes están fuertemente acoplados. Existe una fuerte dependencia entre módulos que puede imposibilitar el desarrollo de funcionalidades nuevas o la corrección de errores sin tener que cambiar el sistema entero.

- **Escalabilidad limitada:** La manera de escalar estas aplicaciones son verticalmente, esto es, agregar más recursos de procesamiento en los servidores de las aplicaciones. Esto lo hace menos viable debido a sus costos y capacidad con respecto al escalado en horizontal.

- **Mantenimiento y evolución compleja:** Con el paso del tiempo, estas aplicaciones se vuelven difíciles de mantener. Además, para agregar nuevas funcionalidades es requerido que los programadores conozcan bien el programa entero debido a su fuerte acoplamiento, lo que puede disminuir la productividad y la fiabilidad del programa por el riesgo de introducir errores.

- **Eficiencia y simplicidad inicial:** En los primeros días de desarrollo, se puede implementar y poner en funcionamiento una aplicación monolítica con bastante facilidad y en poco tiempo. Sin embargo, con el crecimiento de la aplicación y el paso del tiempo, toda la simplicidad inicial se pierde.

2.3.2 Estrategias para Desacoplar Aplicaciones Monolíticas

Antes de empezar, es importante evaluar que aplicaciones monolíticas deberían descomponerse. Hay que descomponer monolitos que tengan problemas de fiabilidad o rendimiento, o que tengan una arquitectura fuertemente acoplada. Es importante también conocer la aplicación, conocer las tecnologías que usa y cuál es su uso u objetivo de negocio [9].

A continuación, se presentan los patrones usados.

2.3.2.1 Descomposición por departamentos o áreas

Es posible usar la organización por departamentos o áreas para descomponer los monolitos en microservicios. Generalmente las organizaciones se dividen en departamentos, por ejemplo, departamento de ventas, departamento de marketing o departamento de atención al cliente [9].

- **Ventajas:** Los equipos de trabajo estarán bien organizados por el departamento y microservicio al que pertenezcan. Esto genera una arquitectura de microservicios estable en el que se encuentran débilmente acoplados [9].
- **Desventajas:** Se requiere un conocimiento acerca de la organización para entender sus divisiones y, además, el diseño de la arquitectura estaría fuertemente acoplado con su división departamental [9].

2.3.2.2 Descomposición por subdominio

Este patrón aprovecha el diseño basado en dominios [10] para descomponer los monolitos según subdominios. Lo que se hace es aprovechar el límite bien definido que existe entre los módulos relacionados con los subdominios para empaquetar dichos módulos en nuevos microservicios [9].

- **Ventajas:** Se vuelven sistemas más escalables y que tienen una arquitectura débilmente acoplada [9].
- **Desventajas:** Se necesita un buen conocimiento acerca de la organización para poder identificar los subdominios. Por otro lado, puede generar demasiados microservicios, volviendo difícil integrarlos [9].

2.3.2.3 Descomposición por transacciones u operaciones

Este patrón descompone los monolitos según la operación o transacción que se deba realizar, esto quiere decir que, las aplicaciones suelen tener que llamar a varios microservicios para realizar una operación o transacción, por lo cual, agrupando los microservicios necesarios para una transacción se puede reducir la latencia y el número de llamadas necesarias. Es adecuado si al agrupar dichos microservicios no formas un monolito [9].

- **Ventajas:** Mejora sobre la consistencia de los datos, los tiempos de respuesta y la disponibilidad [9].
- **Desventajas:** Existe un riesgo de agrupar microservicios que formen un monolito, así como de aumentar la complejidad del microservicio [9].

2.3.2.4 Servicio por equipo de trabajo

Este patrón, como su nombre lo indica, descompone los monolitos en microservicios por equipos de trabajo en lugar de por departamentos, es decir, asigna se crea un microservicio que será totalmente gestionado por cada equipo de trabajo [9].

- **Ventajas:** Cada equipo puede gestionar su microservicio con la tecnología que prefiera, actuando independientemente. Permite un rápido avance e innovación en las funcionalidades del producto [9].
- **Desventajas:** Puede resultar difícil cuando existen dependencias circulares entre equipos [9].

2.3.2.5 Patrón "Strangler fig"

Este patrón envuelve la aplicación monolítica entera en un nuevo sistema, el cual se va incrementalmente transformando en microservicios. Está pensado para sistemas grandes y antiguos, que quieran ser modernizados y permita la coexistencia del sistema antiguo con los nuevos microservicios modernos que se van implementando [9].

- **Ventajas:** Permite agregar nuevos servicios mientras se siguen refactorizando los antiguos, así como también permite la interacción con servicios antiguos que no son ni serán actualizados. Por último, tiene la capacidad de mantener el servicio antiguo en línea mientras se refactoriza el código para actualizarlo [9].
- **Desventajas:** No es posible usarlo en sistemas donde no se puedan interceptar las solicitudes al backend, así como tampoco debería usarse en sistemas pequeños con baja complejidad [9].

2.3.2.6 Patrón de ramificación por abstracción

Este patrón se usa cuando no es posible interceptar las llamadas hacia el monolito y quieres modernizar componentes muy arraigados profundamente en el sistema, con dependencias fuertes [9]. En el siguiente ejemplo se puede entender mejor su funcionamiento.

Imagina que tienes un módulo de procesamientos de pagos que depende de otros componentes, facturación y gestión de pedidos. Necesitas crear una interfaz que defina las interacciones entre el módulo de pagos y sus módulos clientes, facturación y gestión de pedidos. Luego, modificas el módulo de facturación y gestión de pedidos para que usen la interfaz en lugar del módulo de pagos directamente. Por último, desarrollas la nueva implementación del módulo de pagos, con tecnologías modernas y escalables, y cambias la interfaz por el nuevo módulo de pagos.

- **Ventajas:** Permite la coexistencia de múltiples implementaciones de un módulo de software, cambios incrementales reversibles y modernizar funcionalidad que está muy dentro del sistema y a la cual no puedes interceptar sus llamadas [9].
- **Desventajas:** El esfuerzo puede no valer la pena debido a tener que reestructurar demasiado, y a sistemas pobremente estructurados [9].

2.4 Despliegue Blue/Green

Cuando se habla de despliegues blue/green, se habla de un modelo de lanzamiento en el que el tráfico se va redirigiendo gradualmente de una aplicación o microservicio en producción hacia una nueva versión de esta.

La versión blue, o azul, es la versión actual o previa. En cambio, la versión green, o verde, es la versión nueva, a la cual será redirigida todo el tráfico. Cuando todo el tráfico ha sido completamente redirigido, la versión blue se puede conservar en caso de que sea necesario volver a la versión anterior [11].

El uso de implementaciones blue/green traen consigo una serie de ventajas:

- Disminución del tiempo de inactividad.
- Disminución del riesgo de errores.
- Posibilidad de revertir la versión fácilmente en caso de problemas.
- Facilidad para la realización de pruebas.

De estas ventajas se benefician específicamente las aplicaciones críticas, donde el tiempo de inactividad deber ser mínimo. Además, también se benefician los entornos donde las actualizaciones de software son frecuentes y deben ser gestionadas sin grandes interrupciones.

3 Metodología

3.1 Enfoque General del Proyecto

Se ha optado por seguir un enfoque ágil para llevar a cabo este proyecto. Este tipo de metodología permite una gestión de proyectos flexible y reactiva que se ajusta según sea necesario. Esto permite amoldarse a posibles cambios o problemas que puedan surgir. Con este tipo de proyectos donde es necesario integrar varias tecnologías y servicios, la metodología ágil funciona muy bien.

3.2 Herramientas y Tecnologías Utilizadas

La aplicación web facilitada por Amazon que será desplegada como una infraestructura serverless, usa Node.js junto con Express y MustacheExpress. Para su transformación a estructura serverless, se ha decidido usar Vue.js para el frontend de la aplicación. Esta decisión ha sido tomada debido a que Vue.js permite el renderizado en el lado del cliente, facilitando así el despliegue en algún servicio relacionado con infraestructuras serverless.

Amazon Web Services (AWS), es sin duda, la herramienta principal y que da vida a este proyecto. La creación total de la infraestructura fue hecha con los servicios que AWS proporciona, y principalmente, el foco está puesto sobre AWS Lambda. AWS Lambda es uno de los principales servicios cuando hablamos de serverless. Más adelante, en el capítulo de Diseño de la Arquitectura, se expondrá con detalle cuales fueron los servicios específicos usados para construir la infraestructura en su totalidad.

Es importante mencionar que, al principio de este proyecto, mi conocimiento sobre AWS era equivalente al nivel de AWS Cloud Practitioner. Por lo cual, fue necesario adquirir conocimiento sobre los nuevos servicios necesarios para el desarrollo del proyecto.

Para poder trabajar con AWS existen varias maneras de hacerlo:

- **Consola de Administración de AWS:** Aplicación web proporcionada por Amazon que permite la gestión de los recursos AWS.

- **AWS CLI (Interfaz de Línea de Comandos):** Herramienta que permite interactuar con los servicios de AWS a través de comandos en tu Shell.

- **AWS SDKs:** Herramientas que permiten a los desarrolladores un acceso programatico a los servicios de AWS mediante APIs en cada lenguaje de programación especifico.

- **AWS CloudShell:** Shell preautenticado al que se puede acceder desde la consola de AWS y que permite interactuar con los servicios de AWS.

Para el desarrollo del proyecto se usó, en algún momento, cada una de las 4 posibilidades de interacción con AWS. Principalmente se utilizó la SDK de Python, AWS CLI y la consola de AWS.

Para poder hacer uso de AWS CLI es necesario configurar las credenciales de la cuenta AWS que usaremos, es decir, la cuenta AWS donde crearemos los recursos. Para la realización de este proyecto, se hace uso de una cuenta personal creada con este fin. Esto se debe a que, a pesar de que Amazon nos ofrece un "sandbox environment", tiene permisos muy limitados. Para configurar AWS CLI con este entorno ofrecido, primero es necesario realizar la instalación de este. Se usará AWS CLI para Windows, además de Visual Studio Code como centro de operaciones. Una vez instalado AWS CLI, se pueden configurar las credenciales mediante el comando *'aws configure'*.

Como se ha mencionado, se utiliza la herramienta VS Code junto con su extensión AWS Toolkit. Además, Git es esencial para el control de versiones.

Por último, se hace uso de AWS Serverless Application Model (AWS SAM), que es un conjunto de herramientas orientadas al desarrollo de aplicaciones serverless. AWS SAM te permite definir plantillas que luego pueden ser ejecutadas por AWS CloudFormation para la automatización de la creación de la infraestructura.

A continuación, se recopila la lista de herramientas y tecnologías utilizadas:

- Vue.js
- Amazon Web Services (AWS)
- Consola de AWS
- AWS CLI
- AWS SDK de Python
- Visual Studio Code
- Git
- AWS SAM

3.3 Planificación del Trabajo

3.3.1 Lista de Tareas

Se ha definido una lista de tareas junto con su dedicación horaria para la realización del proyecto.

- Estudiar servicios cloud AWS de CI/CD (20h)
- Planificar el diseño de la arquitectura y estimar los costes (20h)
- Analizar la infraestructura de la aplicación monolítica (20h)
- Crear un entorno de desarrollo y un repositorio Git (2h)
- Desacoplar la aplicación monolítica en funciones (60h)
- Migrar la base de datos (2h)
- Automatizar la creación de infraestructura y servicios AWS (76h)
- Definir el pipeline CI/CD (50h)
- Documentación del proyecto (47h)

3.3.2 Diagrama de Gantt

A continuación, en la Figura 1, se puede apreciar el diagrama de Gantt final de este proyecto.

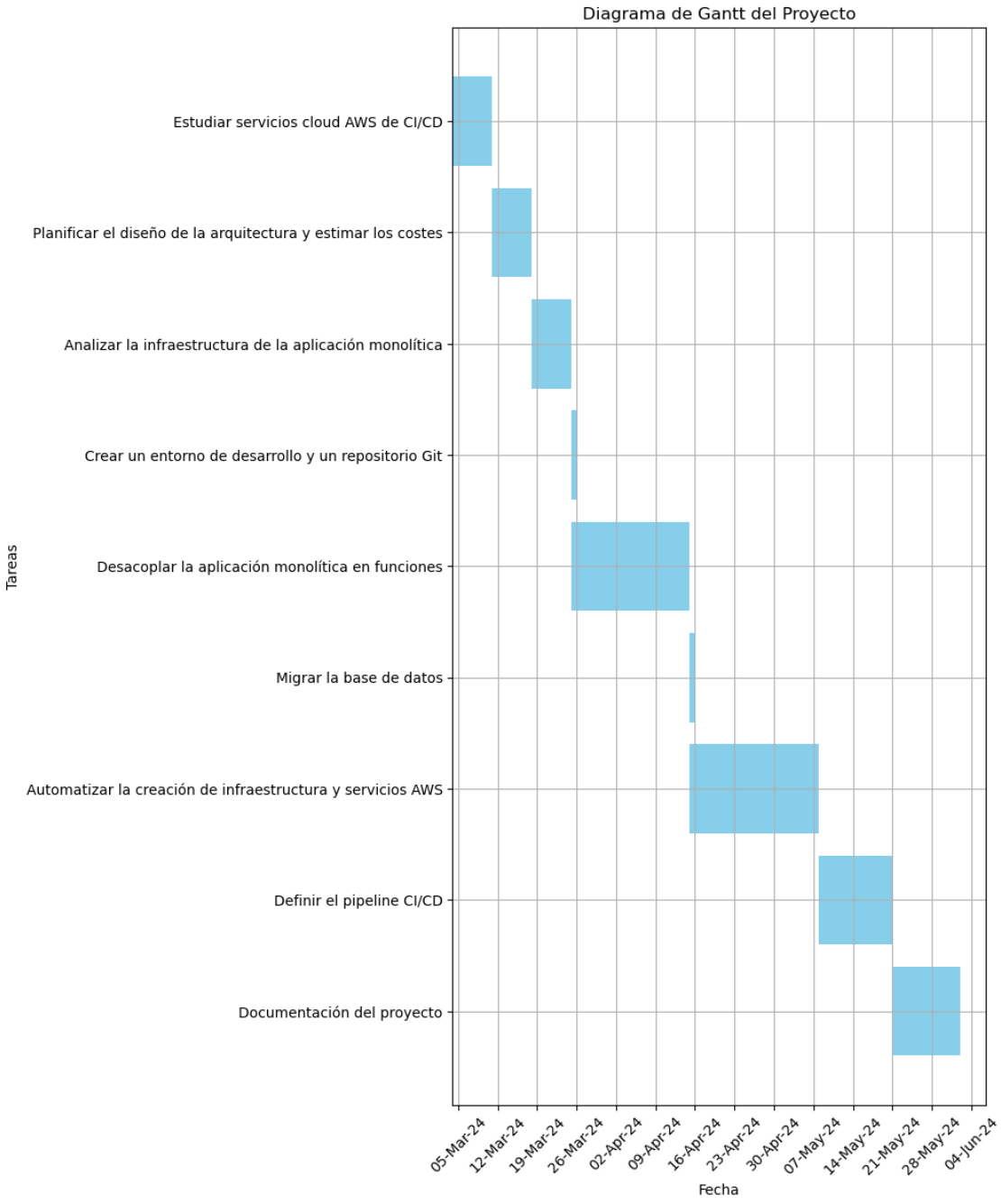

Figura 1. Gráfico de Gantt del Proyecto

4 Diseño de la Arquitectura

4.1 Análisis de la Aplicación Monolítica Existente

Para poder transformar la aplicación monolítica a una arquitectura serverless lo primero que hay que hacer es conocer la aplicación, entender sus funcionalidades y analizar cómo está estructurada, para luego poder diseñar la nueva arquitectura serverless.

La aplicación monolítica facilitada por Amazon para realizar este proyecto se encuentra desplegada en AWS, en un entorno de laboratorio para poder acceder a la aplicación y probar su funcionalidad. Además, allí se encuentra el código fuente de la aplicación.

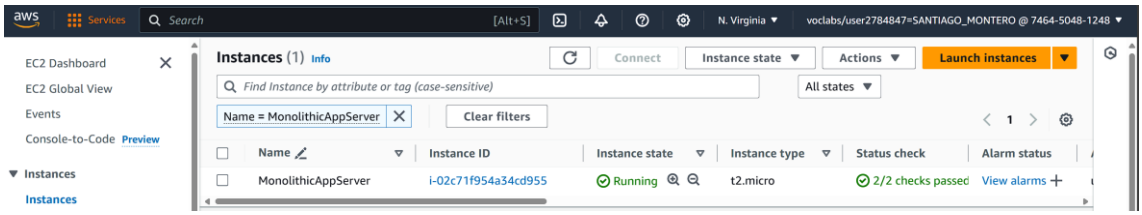

Figura 2. Consola AWS | EC2 Dashboard

Como se puede comprobar en la Figura 2, existe una instancia EC2 iniciada con el nombre de MonolithicAppServer.

En la Figura 3, se muestra el aspecto que tiene la aplicación web.

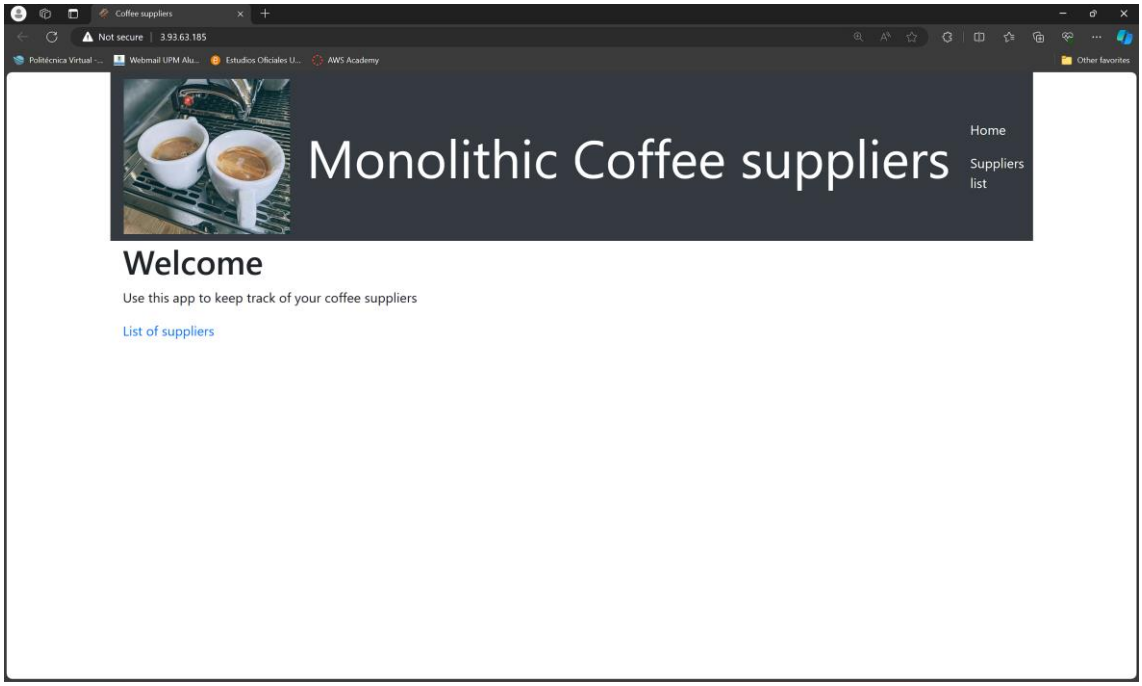

Figura 3. Aplicación web monolítica

La aplicación es bastante sencilla como se puede apreciar, y según el contexto ofrecido por Amazon, se trata de una aplicación que pertenece a una corporación de cafeterías con muchas franquicias que se está expandiendo constantemente y volviéndose muy popular. Es una aplicación de listado de proveedores que tiene problemas de rendimiento y fiabilidad.

La aplicación contiene la página de inicio, que se muestra en la Figura 3, y una página donde se lista a los proveedores junto con su información.

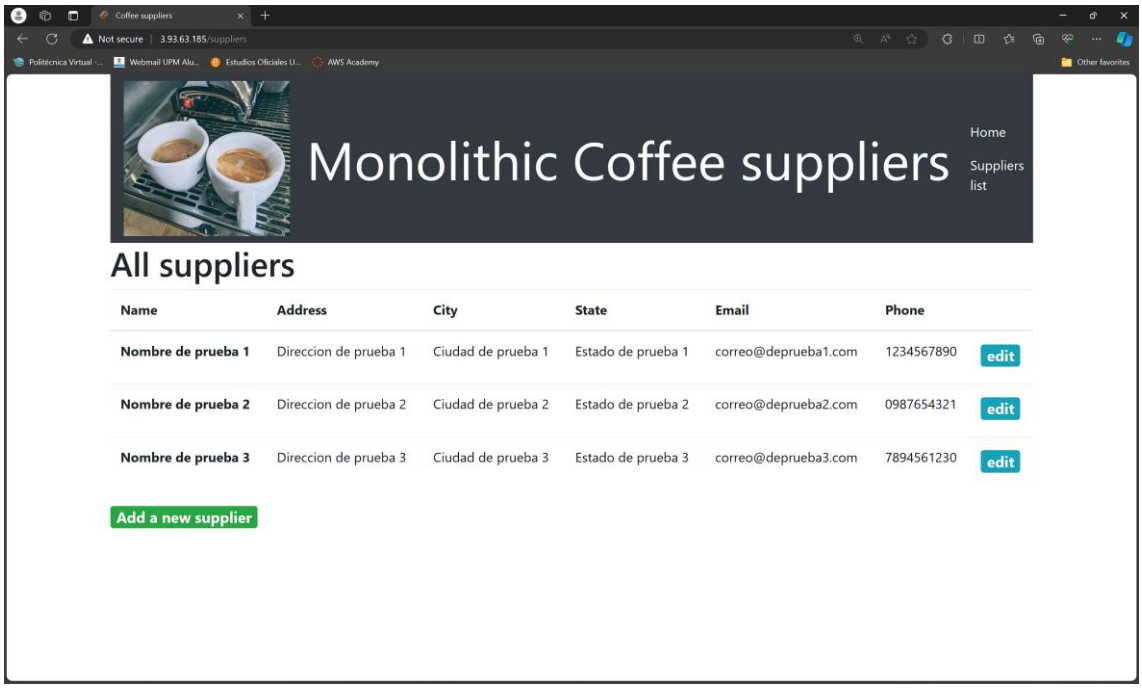

Figura 4. Página de listado de proveedores de la aplicación web

En la Figura 4, se aprecia la página en donde se listan los proveedores, donde además se puede observar que existen botones para agregar y editar la información de cada proveedor. Los botones nos llevan a un formulario que permiten también eliminar el proveedor

La aplicación, evidentemente, tiene persistencia. La instancia EC2 está conectada a una instancia de Amazon RDS, en donde guarda la información de los proveedores en una base de datos MySQL llamada COFFEE.

Esta base de datos solo contiene una tabla sin entradas llamada suppliers.

```
mysql> show full tables from COFFEE;
+------------------+------------+
| Tables_in_COFFEE | Table_type |
+------------------+------------+
| suppliers        | BASE TABLE |
+------------------+------------+
1 row in set (0.00 sec)
```

Figura 5. Listado de tablas de la base de datos COFFEE

4.2 Diseño de la Nueva Arquitectura Serverless

Con el análisis hecho, podemos ver que la aplicación corre en una instancia EC2, es decir una máquina virtual, con 1GB de RAM y 1 vCPU (CPU virtual). Esto hace que se pueda generar un cuello de botella y que sea una solución no escalable, confirmando los problemas mencionados por Amazon: fiabilidad y rendimiento.

Se va a separar la aplicación en backend y frontend, en donde el frontend realice llamadas al backend a través de una API. Esta solución permite el uso de servicios serverless que formaran parte de la nueva arquitectura, y que se muestran a continuación.

4.2.1 Selección de Servicios AWS

A continuación, se listan los servicios AWS que se usaran para la nueva arquitectura, y también, los que forman parte de la solución completa del proyecto.

- **AWS Lambda:** Servicio para la implementación de las funciones que formaran el backend, por ejemplo, guardar proveedores, recuperarlos de la base de datos, etc.
- **API Gateway:** Servicio para la creación de una API que permita la comunicación bidireccional entre frontend y backend. También se encarga de redirigir las solicitudes a las correspondientes funciones lambda.
- **DynamoDB:** Servicio para la base de datos NoSQL serverless que funciona como almacenamiento de los datos de la aplicación de manera escalable y con alta disponibilidad.
- **Amazon S3:** Servicio que permite la creación de un bucket que almacenará los archivos estáticos necesarios para el frontend.
- **AWS CloudFront:** Servicio para la distribución de contenido web estático que tiene como origen el bucket de S3 con el frontend de la aplicación.
- **AWS CodeCommit:** Servicio para la creación del repositorio de la aplicación.
- **AWS CodeBuild:** Servicio para la construcción y despliegue automatizado de la aplicación.
- **AWS CodePipeline:** Servicio para la creación del pipeline CI/CD
- **AWS Identity and Access Management (IAM):** Servicio para la gestión de roles.
- **Amazon CloudWatch Events:** Servicio para la creación de reglas que funcionan como disparadores, en este caso, de la activación del pipeline.
- **AWS CloudFormation:** Servicio para la gestión de infraestructura como código (IaC), que permitirá ejecutar las plantillas de SAM para la creación de la arquitectura serverless y el pipeline.

4.2.2 Estimación de Costos

Para la estimación de costos es necesario hacer algunas suposiciones, ya que no tenemos datos reales de uso de la aplicación. Además, se tiene que considerar el crecimiento continuo de la aplicación y hacer una estimación de los posibles números de uso a los que la aplicación puede llegar.

Vamos a suponer que existen unas 300 franquicias, las cuales visitan la aplicación web varias veces al día. En concreto, vamos a suponer que cada franquicia realiza un promedio de 12 solicitudes (listado y manipulación) por día, y que para cada solicitud son necesarias en promedio 5 solicitudes HTTPS.

A continuación, se muestra la configuración de cada servicio para la estimación de costos.

- **AWS Lambda**
  - **Región:** US East (N. Virginia)
  - **Arquitectura:** x86
  - **Cantidad de solicitudes por mes:** 108.000
  - **Duración de cada solicitud (en ms):** 200
  - **Cantidad de memoria asignada:** 128 MB
  - **Cantidad de almacenamiento efímero asignado:** 512 MB
- **API Gateway**
  - **Región:** US East (N. Virginia)
  - **API de REST**
    - **Unidades de solicitud de la API REST:** miles
    - **Solicitudes por mes:** 108
- **DynamoDB**
  - **Región:** US East (N. Virginia)
  - **DynamoDB capacidad aprovisionada:** true
  - **Clase de tabla:** Estándar
  - **Tamaño del almacenamiento de datos:** 5 GB (Free Tier)
  - **Porcentaje de escrituras no transaccionales:** 100%
  - **Tasa de escritura de referencia:** 2 por segundo (Free Tier)
  - **Tasa de escritura máxima:** 2 por segundo (Free Tier)
  - **Porcentaje de lecturas altamente consistentes:** 100%
  - **Tasa de lectura de referencia:** 2 por segundo (Free Tier)
  - **Tasa de lectura máxima:** 2 por segundo (Free Tier)
- **Amazon S3**
  - **Región:** US East (N. Virginia)
  - **S3 Standard:** true
  - **Data transfer:** true
  - **Almacenamiento de S3 Estándar:** 1MB
  - **Solicitudes PUT a S3 por mes:** 10 (Free Tier)
  - **Solicitudes GET, SELECT por mes:** 20000 (Free Tier) + 61000
  - **Transferencia de datos a:** Amazon CloudFront (Free Tier)
- **Amazon CloudFront**
  - **Región:** US East (N. Virginia)
  - **Estados Unidos, Canadá, Europa:**
    - **Transferencia de datos salientes a Internet por mes:** 5,4 GB (Free Tier)
    - **Número de solicitudes HTTPS por mes:** 540000

- **AWS CodeCommit**
  - **Número de usuarios activos:** 3
- **AWS CodeBuild**
  - **Numero de compilaciones en un mes:** 10
  - **Duración media de la compilación (minutos):** 1
  - **Sistema operativo:** Linux
  - **Tipo de instancia de computación:** general1.small
- **AWS CodePipeline**
  - **Número de pipelines activos utilizados por cuenta al mes:** 1

Con las estimaciones iniciales, y unas pocas más que se hicieron a lo largo de la estimación por servicio, podemos dar una aproximación del costo durante los primeros 12 meses de la arquitectura desplegada en Amazon. Esto es solo una estimación, y no se incluyen los impuestos que puedan aplicar. El costo real podría variar por cuestiones impredecibles. La estimación aprovecha las ventajas de la capa Free Tier de Amazon, la cual te ofrece ciertos servicios gratis para toda la vida y otros solo durante los primeros 12 meses.

La estimación queda como se puede apreciar en la Figura 6 a continuación.

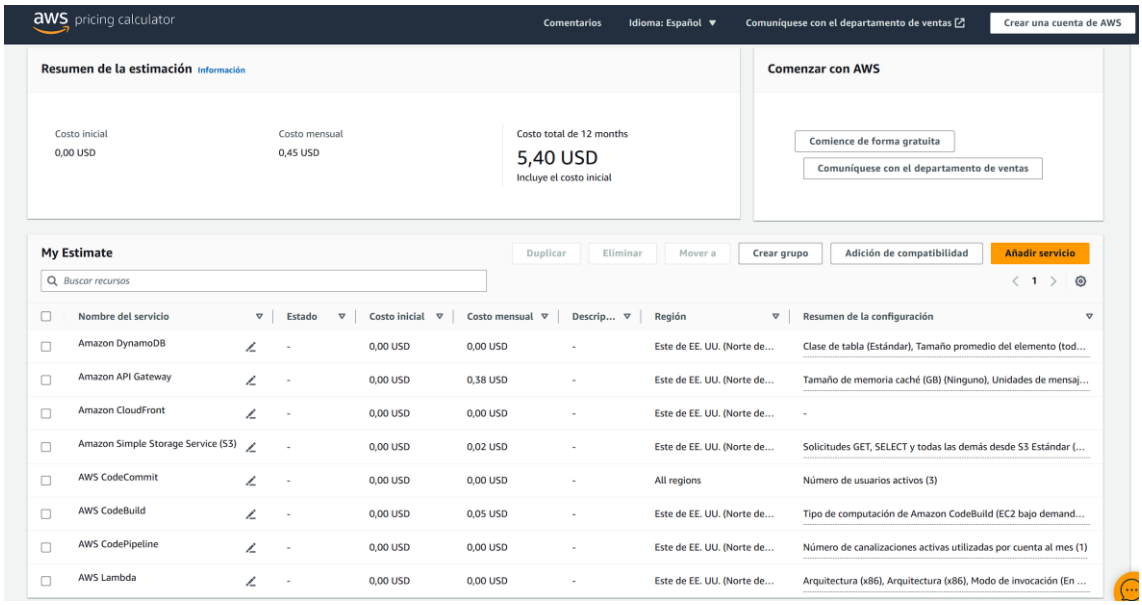

Figura 6. Resumen de la estimación realizada en AWS pricing calculator

Por lo cual, según nuestras suposiciones y la configuración mencionada, el costo de los primeros 12 meses de nuestra arquitectura serverless desplegada en Amazon es asombrosamente del valor de 5,40 USD, al que hay que restarle 4,56 USD gracias al Free Tier de API Gateway durante los primeros 12 meses. Quedando así el aún más asombroso valor de 0,84 USD.

4.2.3 Diagrama de la Arquitectura

Una vez que los servicios AWS que se utilizarán en la arquitectura serverless han sido escogidos, es momento de pasar a la construcción de esta. La nueva arquitectura se puede apreciar en el diagrama a continuación.

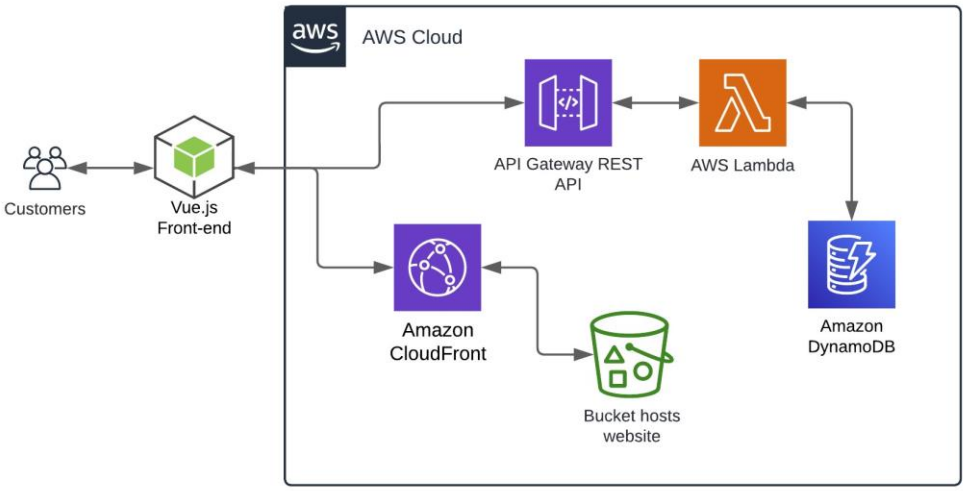

Figura 7. Diagrama de la Arquitectura Serverless

Los clientes (customers) interactuarán con el frontend desarrollado en Vue.js. Las peticiones HTTPS serán servidas por Amazon CloudFront, el cual tiene como origen un bucket de Amazon S3. Dicho bucket contendrá los recursos estáticos de la página web, como los HTML, CSS, Javascript e imágenes. Por otro lado, el backend de la aplicación estará conformado por una REST API. La cual contendrá los siguientes recursos.

- "url-api/suppliers" con métodos GET y POST.
- "url-api/suppliers/{id}" con métodos GET, PUT y DELETE

Por cada uno de los métodos y recursos habrá una función lambda conectada, la cual interactuará con una base de datos NoSQL cumpliendo con la solicitud enviada.

Amazon DynamoDB contendrá una tabla en la que almacenará toda la información de los proveedores. Como dato adicional, los accesos a DynamoDB tienen latencias muy bajas.

Cada servicio de la arquitectura es altamente disponible y cuenta con muy buena escalabilidad.

En la Figura 8, se puede ver el diagrama del pipeline. Los desarrolladores hacen cambios en el repositorio y el pipeline se activa. El código se construye y empaqueta, y después se despliegan los correspondientes cambios en los servicios AWS de la arquitectura serverless.

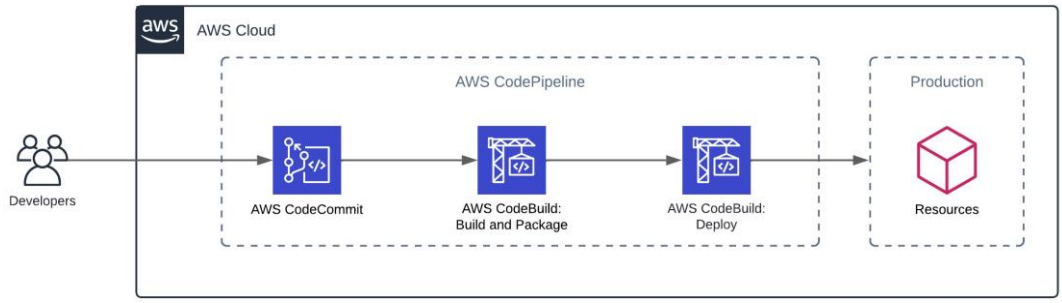

Figura 8. Diagrama del Pipeline

Con la nueva arquitectura diseñada, no solo se solucionan los problemas de escalabilidad, rendimiento y fiabilidad, sino que también se abre la puerta a futuros desarrollos de nuevas funcionalidades con una gran facilidad, ya que solo sería necesario agregar lo nuevo, sin tener que cambiar toda la aplicación. Además, el pipeline facilita el despliegue de la aplicación con su infraestructura, permitiendo productividad y eficiencia.

5 Desarrollo

5.1 Creación del Entorno de Desarrollo

Como se ha mencionado antes, para la realización de este proyecto fue necesario la creación de una cuenta en AWS.

Una vez con la cuenta creada, es hora de acceder a la cuenta y crear un usuario IAM con acceso programático mediante credenciales y permiso para el acceso a la consola de AWS. A este, se le adjunta el rol "AdministratorAccess" administrado por Amazon. Dicho rol permite la gestión completa de todos los servicios de AWS.

Con el usuario IAM creado junto con sus credenciales, se puede pasar a configurar AWS CLI para poder interactuar con AWS a través del comando `'aws configure'`. Las credenciales se tratan de un par de claves: AWS Access Key ID y AWS Secret Access Key. El comando es interactivo, en donde solo es necesario introducir las credenciales.

También es posible acceder a la consola de Amazon, mediante el usuario y contraseña que se definió a la hora de crear el usuario, para tener una interacción con AWS más gráfica.

Por último, hay que obtener el código de la aplicación monolítica. Como se mencionó anteriormente, este código es facilitado por Amazon y se encuentra en la instancia EC2 que está desplegada en el entorno de laboratorio. El código tiene la estructura que se muestra en la Figura 9.

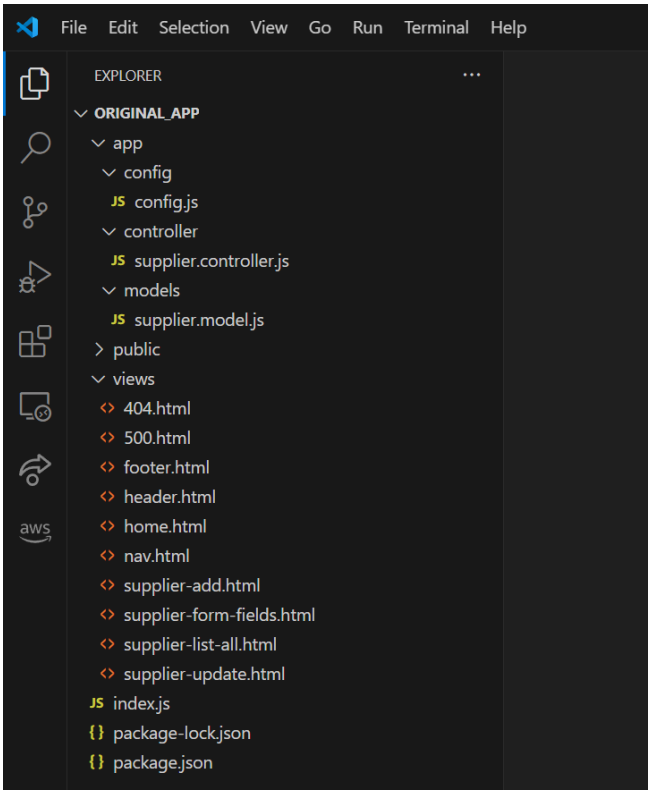

Figura 9. Estructura del código de la aplicación monolítica

5.2 Desacoplamiento de la Aplicación Monolítica

Para empezar con el desacoplamiento de la aplicación, hay que fijarse en cómo funciona el código actual. Para esto, hay que centrarse mayoritariamente en el punto de entrada principal de la aplicación, que es el archivo index.js.

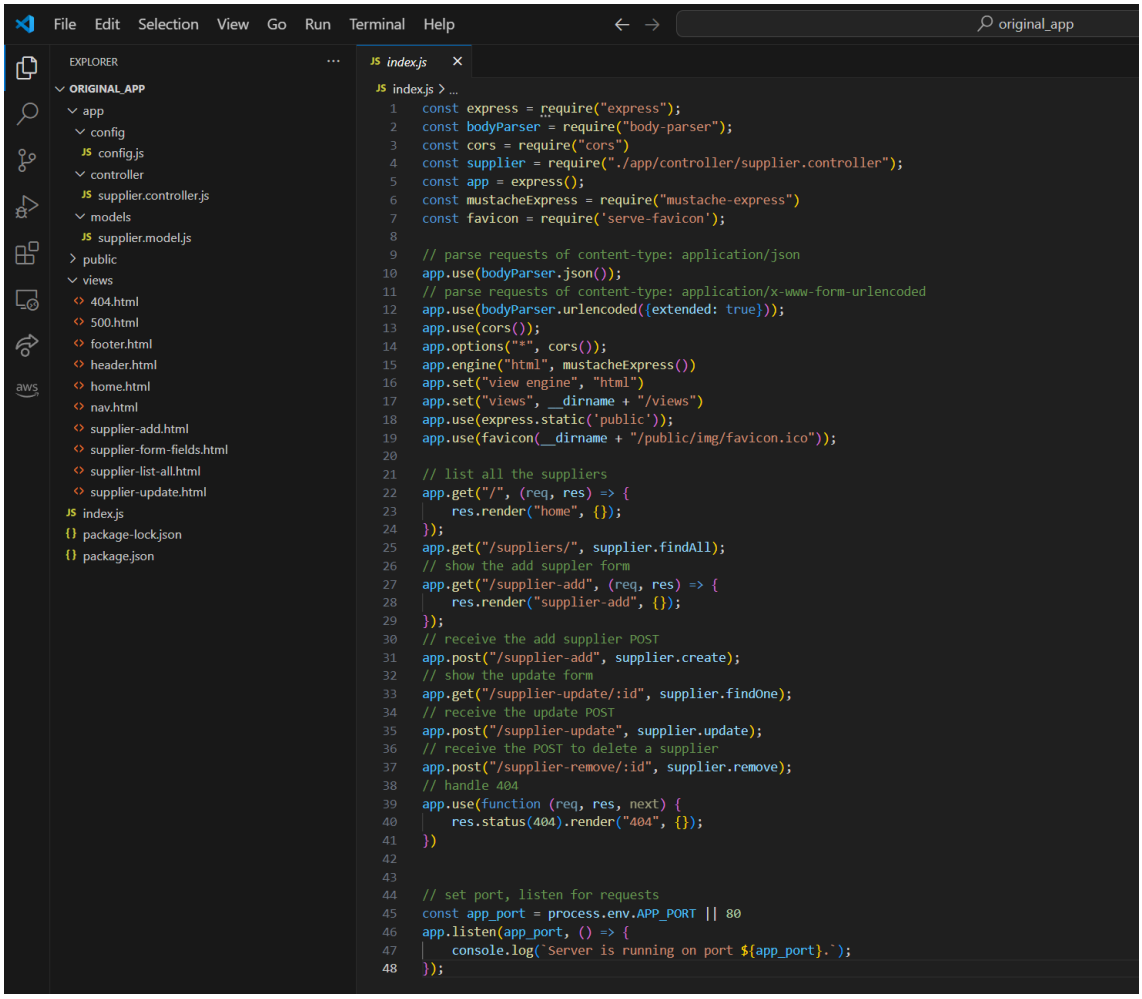

Figura 10. Punto de entrada principal a la aplicación

Con el código mostrado en la Figura 10 se puede extraer casi toda la información necesaria para entender el cómo funciona la aplicación.

Una de las cosas importantes es saber que usa Express.js para la creación de un servidor web, y que, además, usa mustacheExpress para la renderización de HTML.

Sin embargo, lo más importante es fijarse en las rutas que están definidas, las cuales sirven para agregar, listar, actualizar y eliminar los proveedores. Esto es lo más importante ya que aquí se marcará la separación entre el frontend y el backend, es decir, estas rutas serán los recursos de la API que se definirá como parte del backend que será usado por el frontend para realizar las llamadas respectivas.

El archivo config.js sirve para la crear la conexión con la base de datos, y el archivo supplier.model.js define el modelo de datos del proveedor y las funciones para interactuar con la base de datos. De este modo, estos archivos prácticamente no tienen utilidad, ya que para lo único que podían servir era para conocer el modelo de datos del proveedor, sin embargo, este se puede conocer gracias a la base de datos y la estructura de su tabla COFFEE vista anteriormente.

El archivo supplier.controller.js contiene la lógica del controlador para manejar las operaciones realizadas sobre los proveedores, y en qué desencadena cada solicitud. Esto se puede ver como la navegación que existe en la aplicación, lo cual servirá para construir el frontend.

Por último, se encuentra la carpeta de views que contiene los archivos HTML que definen las vistas de la aplicación, lo cual también será útil en la construcción del frontend.

Con este pequeño análisis realizado, se puede concluir que es necesaria la construcción de un frontend, reutilizando los archivos HTML y la lógica del controlador, que realice peticiones al backend a través de una API.

5.2.1 Frontend

Se usará Vue.js para la construcción del frontend debido a su renderización client-side, lo cual se acerca más a un enfoque serverless que tener un servidor web con Express corriendo en una máquina virtual que sería necesario administrar. Este frontend estará alojado en un bucket S3 y será distribuido por Amazon CloudFront.

¿Por qué no solo alojar el frontend en el bucket S3 como web estática? Esto se debe a que con Amazon CloudFront se pueden obtener ventajas significativas frente al uso de solo S3 como web estática. A continuación, se detallan las principales razones por las cuales se opta al uso de CloudFront:

- **Rendimiento:** CloudFront distribuye el contenido a través de sus "Edge Locations", lo que significa que las solicitudes se resuelven desde el servidor más cercano a los clientes. Esto sumado con el caching que realiza resultan en un mejor rendimiento con latencias y tiempos de carga bajos.
- **Seguridad:** CloudFront proporciona HTTPS sin la necesidad de configurar un certificado en S3. Además, tiene protección contra ataque DDoS integrada.
- **Flexibilidad y escalabilidad:** Debido a lo mencionado en el punto de rendimiento, CloudFront distribuye el tráfico uniformemente, lo cual mejora la escalabilidad en momentos de picos de tráfico.

Para construir esta aplicación en Vue.js se reutilizará los archivos contenidos en la carpeta "views" del código original de la aplicación.

Como resultado, en la Figura 11 se puede observar los componentes de Vue.js creados para el nuevo frontend.

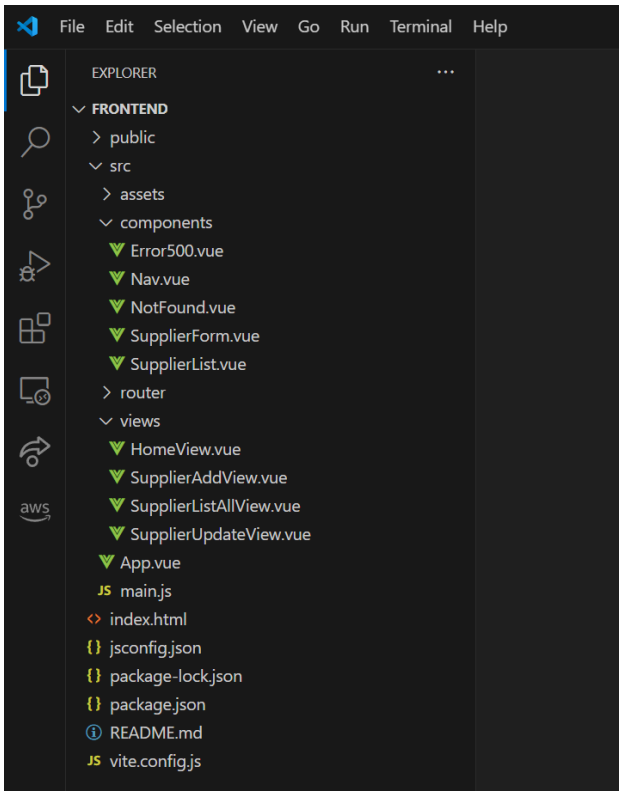

Figura 11. Estructura del código del nuevo frontend

Se crearon las 4 páginas principales que tenía la aplicación original, la página principal, la página donde se listan los proveedores, la página con el formulario para añadir un nuevo proveedor, y la página con el formulario para actualizar o eliminar un proveedor. Los demás componentes sirven para crear estas páginas, como por ejemplo SupplierForm.vue, que es el componente que contiene el formulario común para las páginas donde se muestra el formulario.

Una vez listo y probado localmente el frontend desarrollado con Vue.js, es momento de crear un bucket S3 para alojarlo. Mediante la consola de Amazon se crea un bucket S3 simple y se sube la carpeta resultada de ejecutar `'npm run build'`.

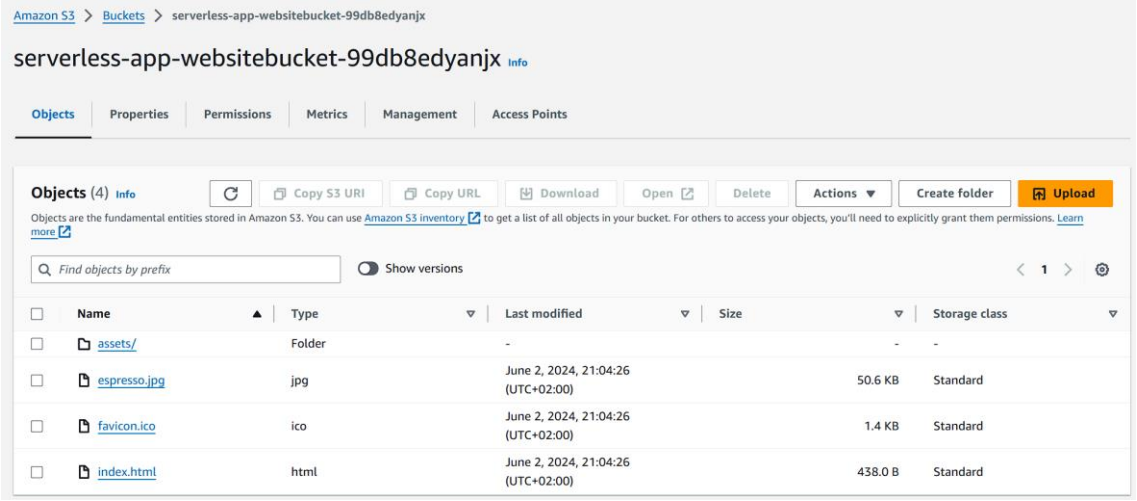

Figura 12. Bucket S3 con los archivos del frontend

Para la creación del bucket se usó las opciones por defecto, a excepción de la habilitación del versionado del bucket.

Para probar el frontend alojado en S3, aunque sin conexión al backend todavía, es necesario crear la distribución CloudFront. Para esto, se crea una distribución de CloudFront en donde se le indica el S3 bucket como origen. Además, se restringe el acceso del bucket S3 para que solo se pueda acceder a través de CloudFront. Esto genera una policy que debemos agregar al bucket. Además, se añade a la configuración que cuando soliciten la página con http, los redirija a https. También se configura la caché y la redirección de errores hacia index.html para que Vue.js lo maneje correctamente a través de su gestor de router.

Una vez que se tiene la distribución creada y configurada, se puede acceder a la aplicación web mediante la URL proporcionada por CloudFront.

Se puede observar en la Figura 13 que, efectivamente, se puede acceder a la página web. Sin embargo, debido a que la API aún no está creada, no es posible conectarse a ella, por lo cual se muestra el mensaje de error.

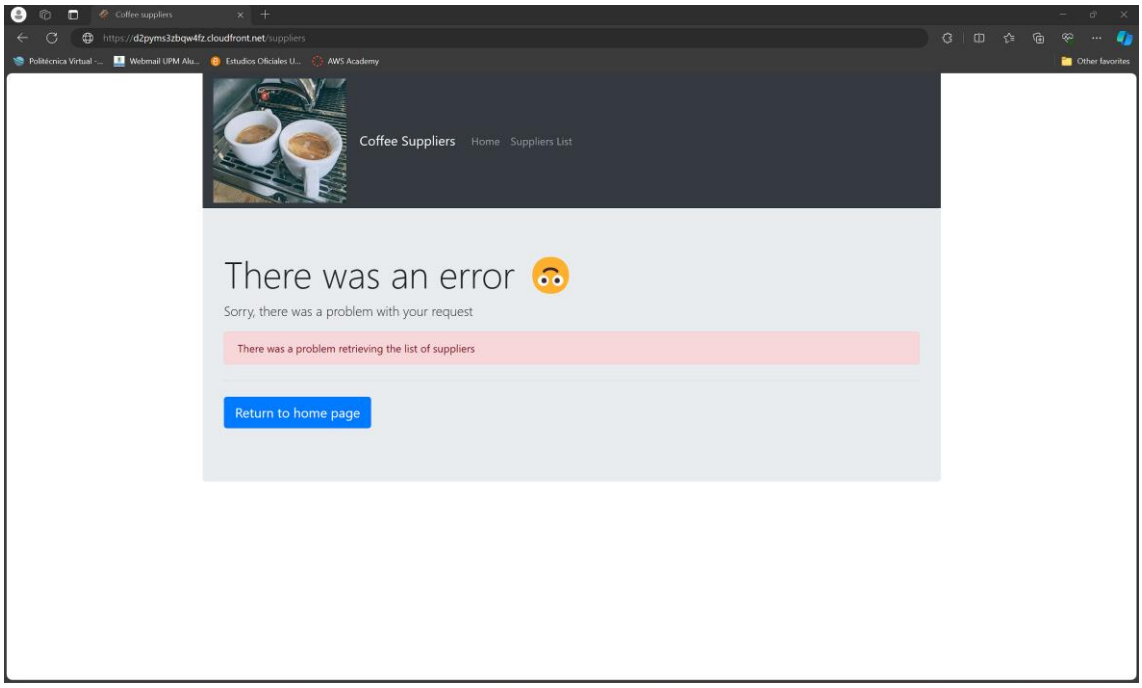

Figura 13. Página web accedida mediante la URL proporcionada por CloudFront

5.2.2 Base de datos

Para sustituir la base de datos usada en la aplicación original se usará DynamoDB, que es una base de datos serverless NoSQL. Aunque existe la posibilidad de crear una base de datos SQL serverless gracias a Aurora, se ha optado por escoger DynamoDB debido a que se apega más a la arquitectura serverless, debido a que es un servicio completamente administrado y altamente escalable. Además, DynamoDB cuenta con la capa gratuita de Amazon por siempre, en donde obtienes 25GB de almacenamiento y 25 unidades de capacidad de escritura y lectura, lo que es suficiente para administrar 200 millones de solicitudes al mes. Por el contrario, Aurora solo ofrece su capa gratuita durante los primeros 12 meses desde la creación de la cuenta AWS, con 750 horas al mes de uso de la base de datos y 20GB de almacenamiento.

Solo es necesario crear una tabla dentro de DynamoDB. DynamoDB guarda los datos en tablas, y dichas tablas guardan ítems, que a su vez almacenan atributos. Se creará una tabla con 2 unidades de capacidad de escritura y 2 de lectura y con el id del proveedor como 'Partition Key', o en este caso, clave primaria.

Con la tabla creada, es posible agregar ítems directamente desde la consola de Amazon, como se muestra en la Figura 14.

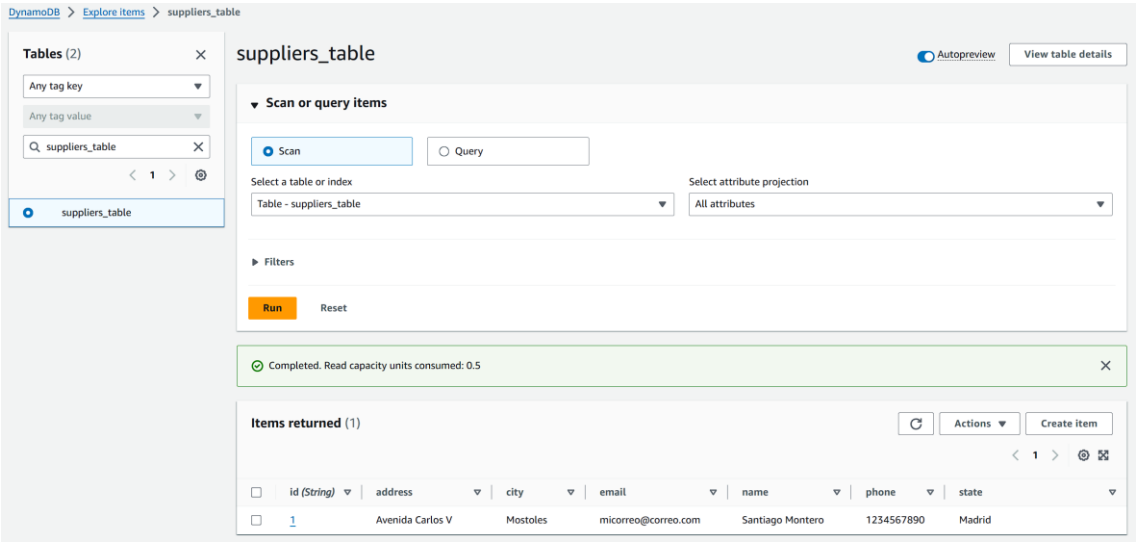

Figura 14. Tabla de DynamoDB con un item creado

5.2.3 Funciones Lambda

Para la creación de las funciones lambdas se tendrá en cuenta las solicitudes que se hacían en la aplicación original a cada una de las rutas. Cada una de estas solicitudes ahora apuntarán a una API, la cual a su vez apuntará a cada una de las funciones lambda para el procesamiento de la solicitud. Por lo tanto, se van a crear 5 funciones lambda que serán escritas en Python.

Cada función obtiene el nombre de la tabla de DynamoDB a través de una variable de entorno denominada TABLE_NAME. Haciendo uso de la AWS SDK para Python, llamada Boto3, se puede interactuar directamente con DynamoDB y hacer las operaciones correspondientes.

Las funciones se crearán en una carpeta con la estructura que se puede observar en la Figura 15, de donde luego se subirán a AWS Lambda para su despliegue y configuración de rol que permita la interacción con DynamoDB.

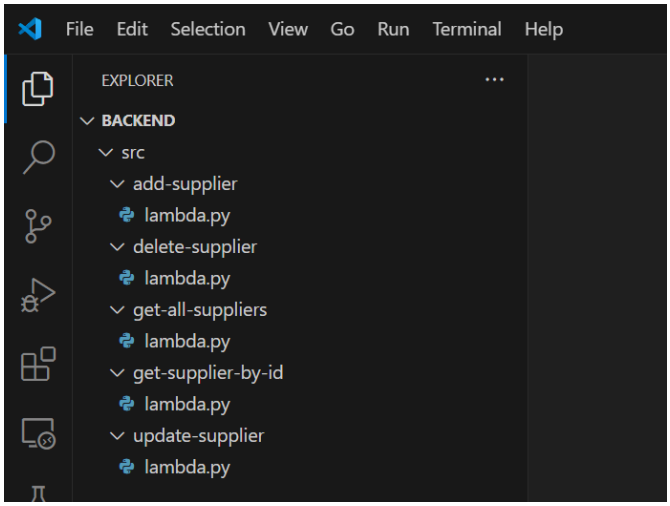

Figura 15. Estructura para la creación de las funciones lambda

Las funciones cumplen la funcionalidad descrita por su nombre.

- **Función add-supplier**

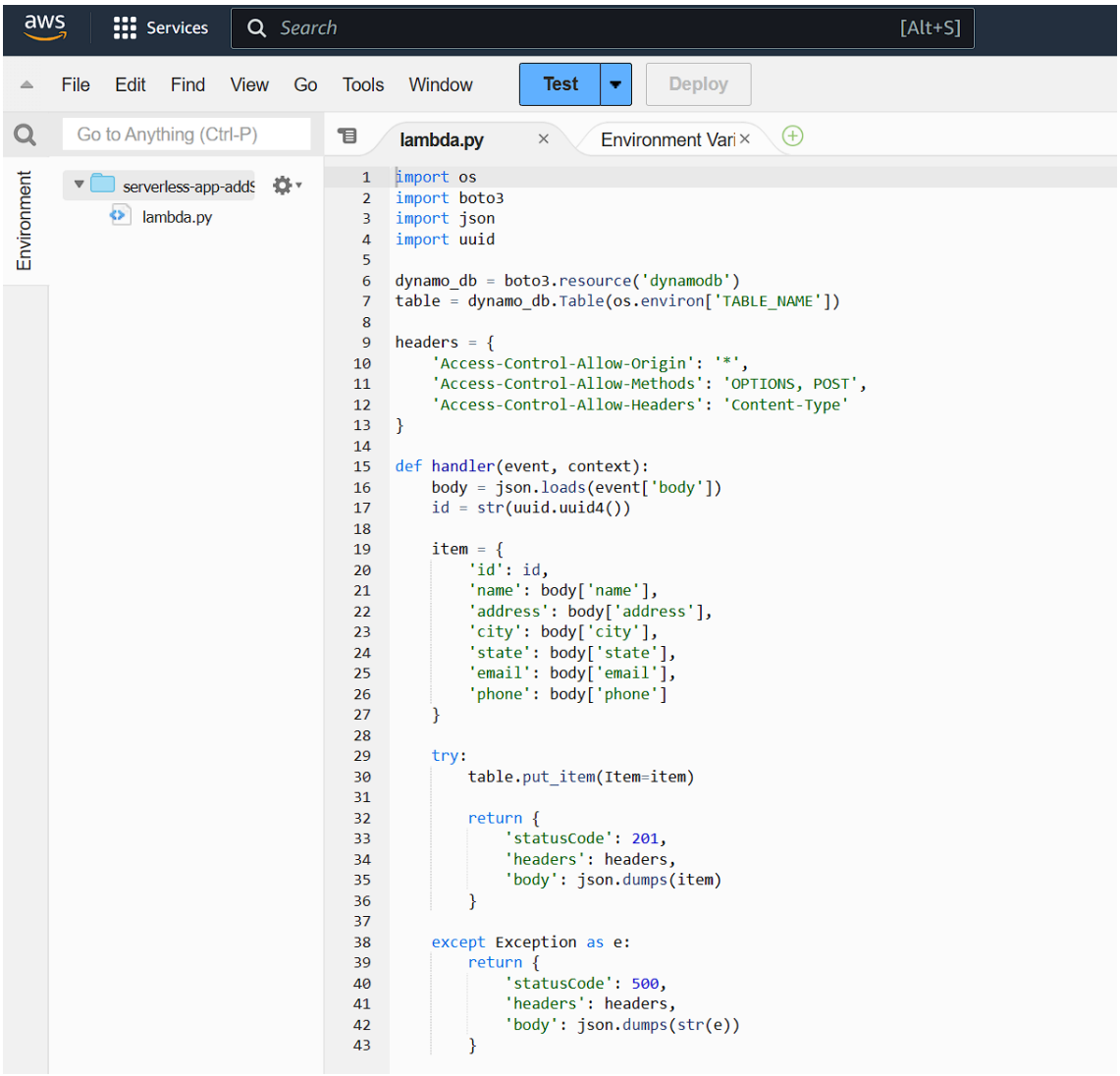

Figura 16. Implementación de función para añadir un proveedor

Esta función lambda se encarga de recibir la información de un nuevo proveedor y crear un nuevo item en la tabla de DynamoDB. Si todo va bien, devuelve el nuevo proveedor creado junto al id generado y un código de respuesta 201.

- **Función delete-supplier**

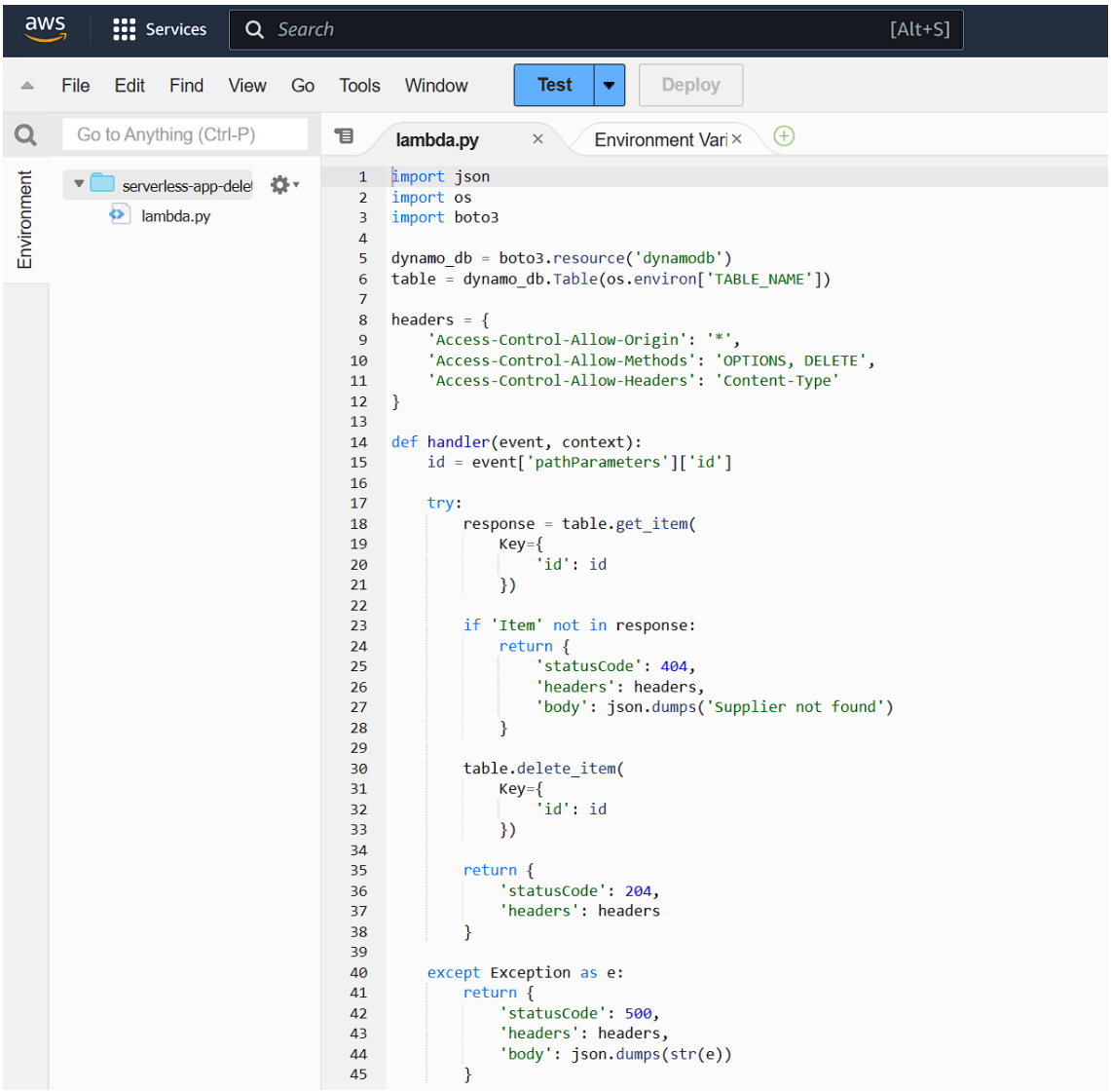

Figura 17. Implementación de función para eliminar un proveedor

Esta función lambda se encarga de eliminar un proveedor de la tabla de DynamoDB según el id pasado en el path de la solicitud. Si todo va bien, devuelve un código de respuesta 204.

- **Función get-supplier-by-id**

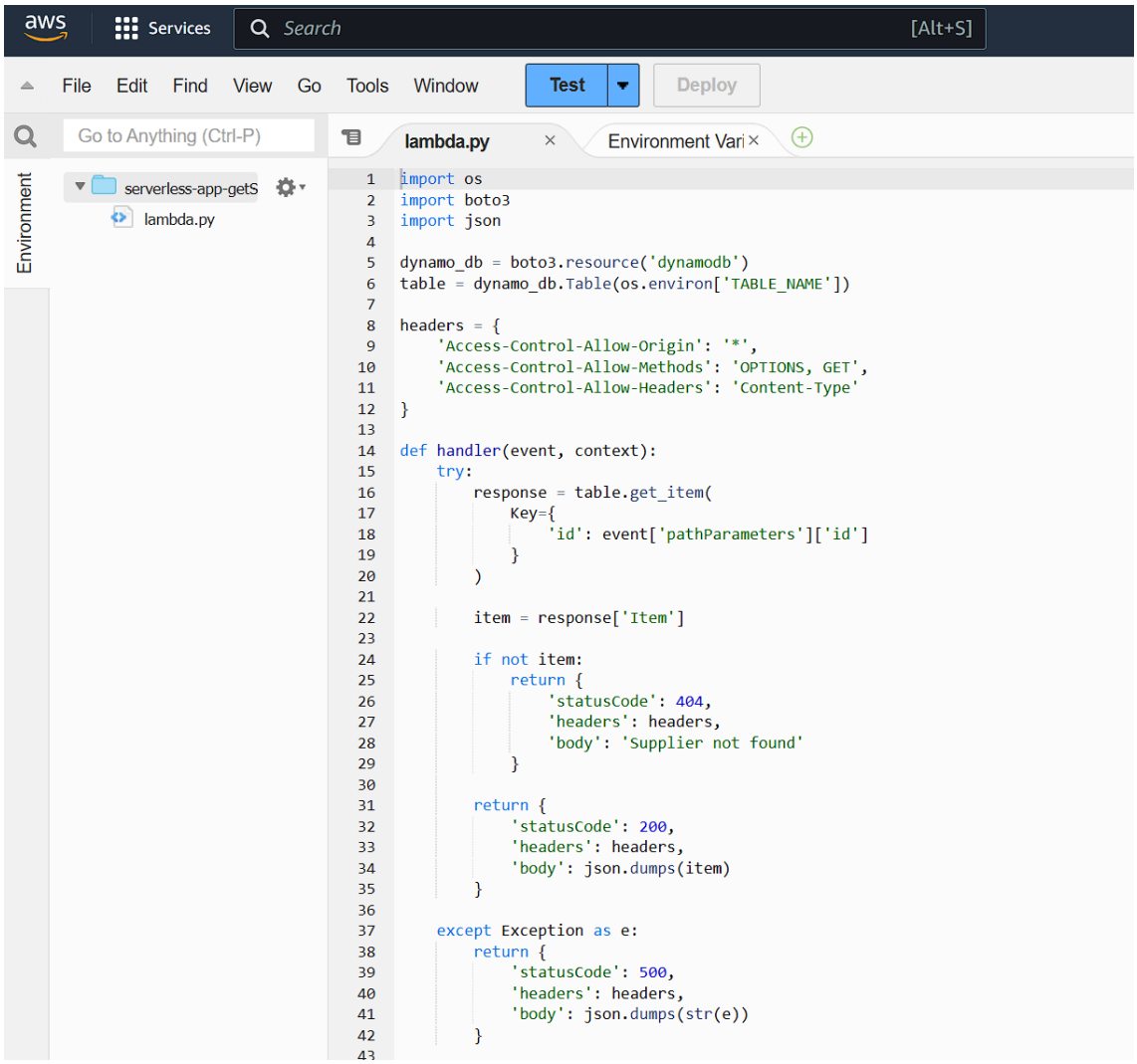

Figura 18. Implementación de función para obtener un proveedor por su id

Esta función lambda se encarga de obtener y devolver la información de un proveedor de la tabla de DynamoDB dado su id en el path de la solicitud. Si todo va bien, devuelve la información del proveedor solicitado junto con un código de respuesta 200.

- **Función update-supplier**

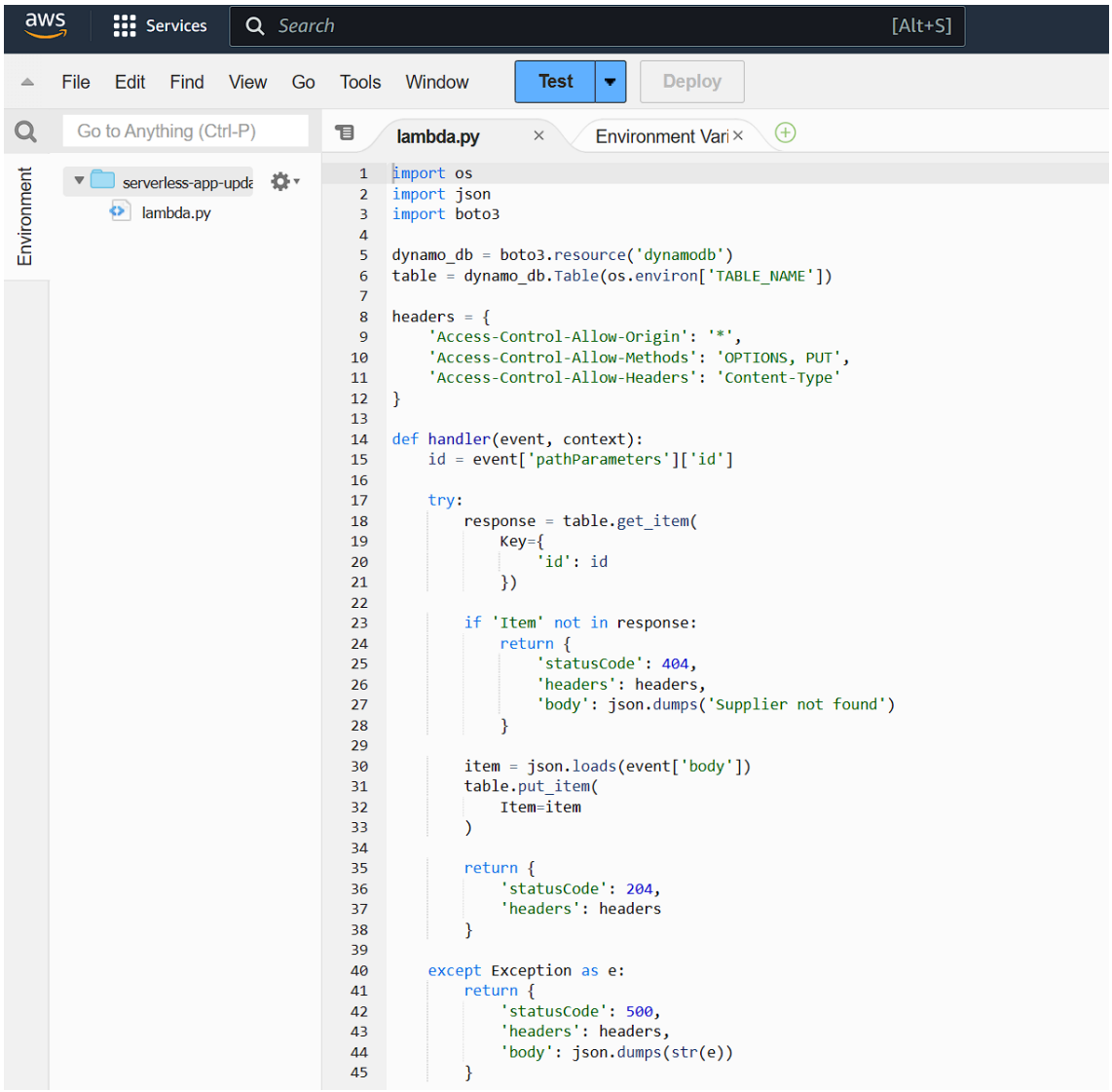

Figura 19. Implementación de función para actualizar un proveedor

Esta función lambda se encarga de actualizar la información de un proveedor en la tabla de DynamoDB con la nueva información pasada a través del cuerpo de la solicitud. Si todo va bien, devuelve un código de respuesta 204.

- **Función get-all-suppliers**

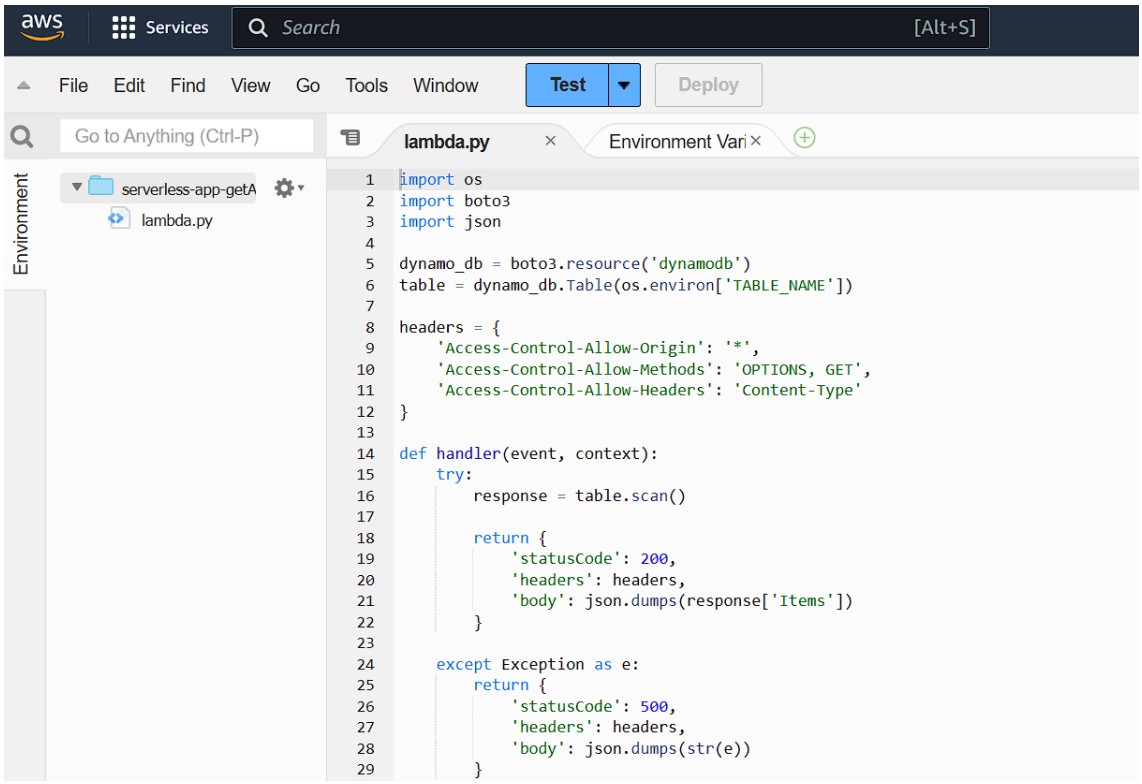

Figura 20. Implementación de función para obtener todos los proveedores

Esta función lambda se encarga de devolver una lista de todos los proveedores, junto con su información, existentes en la tabla de DynamoDB. Si todo va bien, devuelve todos los proveedores y un código de respuesta 200.

5.2.4 API

Para la creación de la REST API, se creará una API de tipo REST en API Gateway. Dentro de esta, se creará un recurso por cada uno de los que aparecen en la aplicación original con sus respectivos métodos, es decir, se crearán los siguientes recursos:

- /suppliers
  - GET
  - POST
  - OPTIONS (para el manejo de solicitudes preflight en CORS)

- /suppliers/{id}
  - DELETE
  - GET
  - PUT
  - OPTIONS (para el manejo de solicitudes preflight en CORS)

Cada recurso creado se conectará con su función lambda correspondiente para gestionar la solicitud. Una vez que cada función lambda está conectada a cada recurso, cada vez que se haga una solicitud a ese recurso, la función lambda será lanzada y se obtendrá una respuesta.

Para probar su funcionamiento se creará un despliegue de la API creada. Con la URL proporcionada por el despliegue de la API y con la ayuda de POSTMAN se comprobará el funcionamiento de esta.

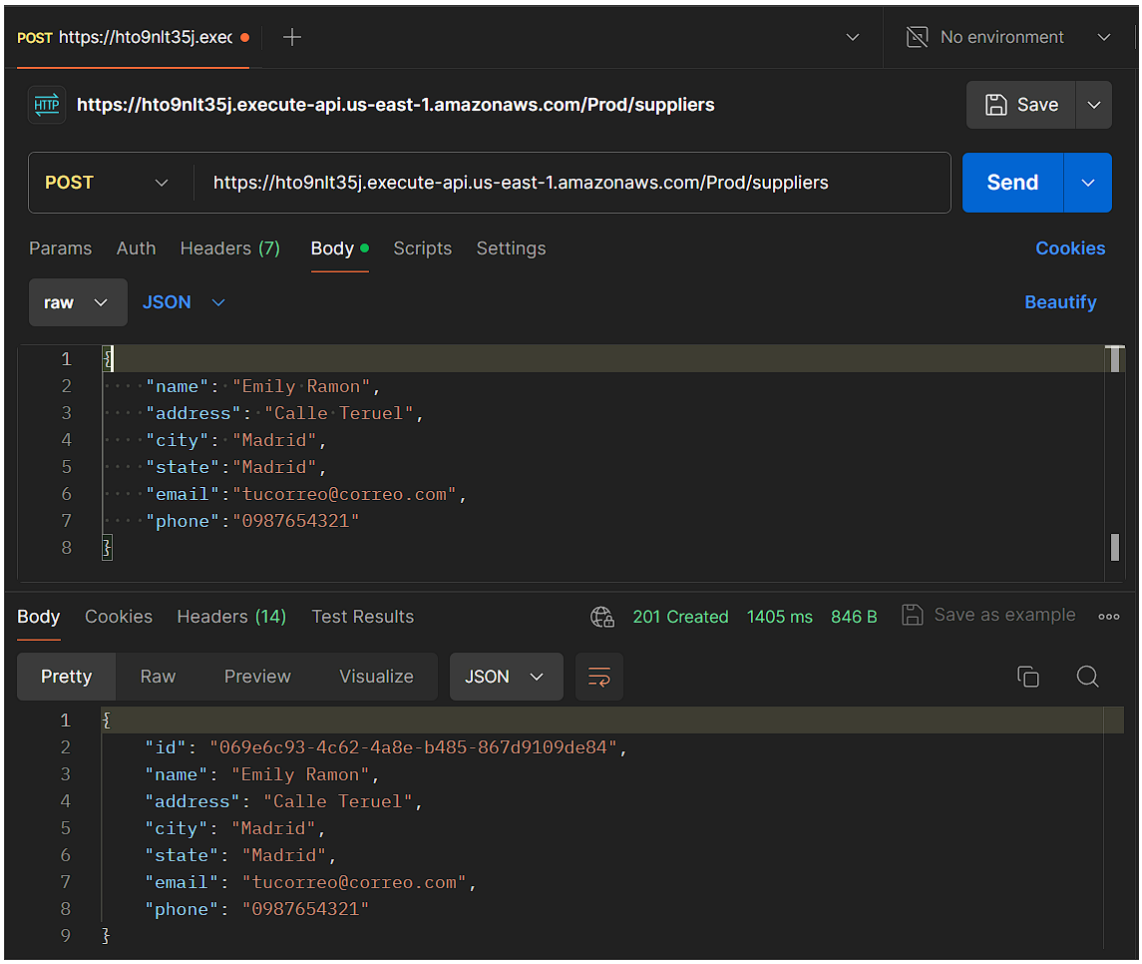

Figura 21. Solicitud POST hacia la API creada

Una vez que se comprueba el funcionamiento de la API, solo faltaría configurar una variable de entorno adecuada en el frontend para que pueda enviar solicitudes hacia la API.

Para actualizar el frontend, es necesario cargar nuevamente los archivos en el bucket S3, pero esta vez agregando un archivo .env, con una variable de entorno que contenga el endpoint de la API, antes de correr `'npm run build'`.

Con el frontend actualizado, finalmente se obtiene la aplicación monolítica entera desacoplada y desplegada en una arquitectura serverless. Accediendo nuevamente a la URL proporcionada por CloudFront se puede acceder a la aplicación web y comprobar que efectivamente el post realizado a través de POSTMAN se hizo efectivo. Es posible también agregar más proveedores, editar su información y eliminarlos.

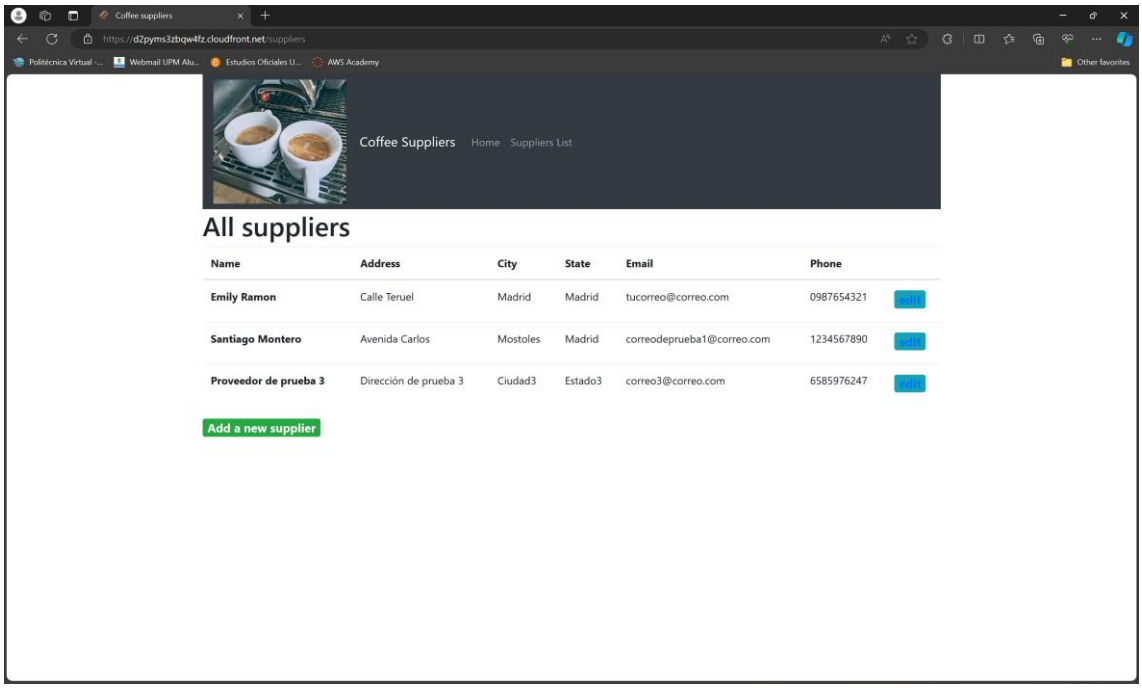

Figura 22. Aplicación web con 3 proveedores de prueba

5.3 Automatización de la Creación de la Infraestructura

Para la automatización de la creación de la infraestructura es necesario definir la infraestructura como código (IaC). Para ello, AWS ofrece herramientas para este fin. En este proyecto se ha optado por el uso de AWS SAM debido a que el desarrollo se trata de una aplicación serverless. AWS SAM permite definir aplicaciones serverless a través de plantillas con formato YAML. En estas plantillas la definición de, por ejemplo, APIs, funciones lambda o bases de datos es más simple debido al uso de una sintaxis simplificada comparándolo con una plantilla de CloudFormation. Sin embargo, al final AWS CloudFormation se encarga de la realización de las tareas necesarias para el despliegue de los servicios definidos en la plantilla de AWS SAM.

Por lo tanto, para lograr la automatización de la creación de la infraestructura, es necesario definir una plantilla SAM con todos los servicios que se han creado en los apartados anteriores y su configuración correspondiente.

El nombre de la plantilla SAM, en este caso, será `'template.yaml'`.

A continuación, se muestra la definición en la plantilla de los servicios creados anteriormente. Se mostrarán colapsados los campos más largos y con menos relevancia.

- Rol que permite la interacción con DynamoDB y logs de CloudWatch a las funciones lambda.

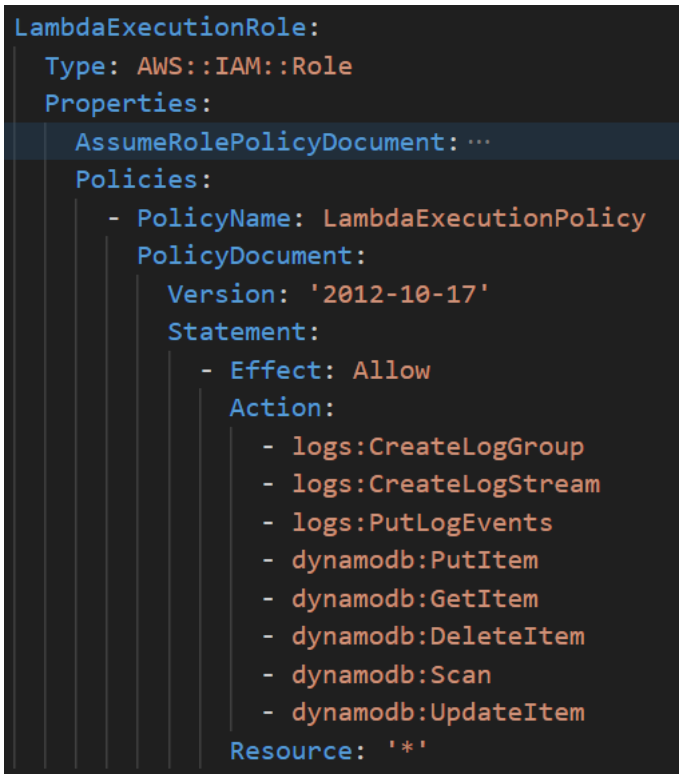

Figura 23. Definición de LambdaExecutionRole

- API de API Gateway con los métodos permitidos y un despliegue con nombre Prod. Más adelante se definen los recursos de esta API mediante la definición de las funciones lambda.

Figura 24. Definición de CoffeeSuppliersApi

- Todas las funciones lambda se definen de la misma manera que la indicada en la Figura 26, en donde asumen el rol creado antes, especifican el origen de su código, se crean sus variables de entorno para el nombre de la tabla de DynamoDB y se define el recurso de la API con el que la función será lanzada a ejecución.

```yaml
addSuplierFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: backend/src/add-supplier/
    Handler: lambda.handler
    Runtime: python3.11
    Role: !GetAtt LambdaExecutionRole.Arn
    Environment:
      Variables:
        TABLE_NAME: !Ref SuppliersTable
    Events:
      Api:
        Type: Api
        Properties:
          Path: /suppliers
          Method: POST
          RestApiId: !Ref CoffeeSuppliersApi
```

Figura 25. Definición de función lambda

- Tabla de DynamoDB para el almacenamiento de la información de los proveedores. Se define id como su clave primaria y se configura las unidades de capacidad de escritura y lectura a 2.

```yaml
SuppliersTable:
  Type: AWS::Serverless::SimpleTable
  Properties:
    PrimaryKey:
      Name: id
      Type: String
    ProvisionedThroughput:
      ReadCapacityUnits: 2
      WriteCapacityUnits: 2
```

Figura 26. Definición de SuppliersTable

- Bucket S3 para el alojamiento de la aplicación web desarrollada en Vue.js junto con su Policy para solo permitir el acceso al bucket mediante CloudFront.

```yaml
WebsiteBucket:
  Type: AWS::S3::Bucket
  DeletionPolicy: Retain
  Properties:
    VersioningConfiguration:
      Status: Enabled

WebsiteBucketPolicy:
  Type: AWS::S3::BucketPolicy
  Properties:
    Bucket: !Ref WebsiteBucket
    PolicyDocument:
      Version: '2012-10-17'
      Id: PolicyForCloudFrontPrivateContent
      Statement:
        - Sid: AllowCloudFrontServicePrincipal
          Effect: Allow
          Principal:
            Service: cloudfront.amazonaws.com
          Action: s3:GetObject
          Resource: !Sub arn:aws:s3:::${WebsiteBucket}/*
          Condition:
            StringEquals:
              AWS:SourceArn: !Sub arn:aws:cloudfront::${AWS::AccountId}:distribution/${CloudFrontDistribution}
```

Figura 27. Definición de WebsiteBucket y su Policy

- Control de acceso de origen para la distribución de CloudFront que permite que solo la distribución de CloudFront tenga acceso al origen, que es el bucket S3.

```yaml
CloudFrontOriginAccessControl:
  Type: AWS::CloudFront::OriginAccessControl
  Properties:
    OriginAccessControlConfig:
      Name: !Sub ${WebsiteBucket} OAC
      OriginAccessControlOriginType: s3
      SigningBehavior: always
      SigningProtocol: sigv4
```

Figura 28. Definición de CloudFrontOriginAccessControl

- Distribución de CloudFront con WebsiteBucket como origen. Se define el objeto por defecto, index.html. Se configura la redirección de errores para que Vue.js pueda gestionar adecuadamente las rutas. Se configura las opciones de caché y la opción para redirigir las solicitudes desde HTTP a HTTPS. Por último, se configura PriceClass que indica que solo se usaran las ubicaciones de borde de Europa, Israel, Turquía, Estados Unidos, México y Canadá.

```yaml
CloudFrontDistribution:
  Type: AWS::CloudFront::Distribution
  Properties:
    DistributionConfig:
      Origins:
        - DomainName: !GetAtt WebsiteBucket.RegionalDomainName
          Id: myS3Origin
          OriginAccessControlId: !GetAtt CloudFrontOriginAccessControl.Id
          S3OriginConfig:
            OriginAccessIdentity: ''
      Enabled: true
      DefaultRootObject: index.html
      CustomErrorResponses:
        - ErrorCode: 403
          ErrorCachingMinTTL: 3600
          ResponseCode: 200
          ResponsePagePath: /index.html
      HttpVersion: http2
      DefaultCacheBehavior:
        AllowedMethods:
          - DELETE
          - GET
          - HEAD
          - OPTIONS
          - PATCH
          - POST
          - PUT
        CachedMethods:
          - GET
          - HEAD
        TargetOriginId: myS3Origin
        ForwardedValues:
          QueryString: false
          Cookies:
            Forward: none
        ViewerProtocolPolicy: redirect-to-https
        MinTTL: 0
        DefaultTTL: 3600
        MaxTTL: 86400
      PriceClass: PriceClass_100
      ViewerCertificate:
          CloudFrontDefaultCertificate: true
```

Figura 29. Definición de CloudFrontDistribution

Con el template creado es posible realizar el despliegue de este, mediante AWS SAM. Para efectuarlo se ejecuta primero `'sam build'.` A continuación, se ejecuta el comando `'sam deploy –guided'`, el cual realiza una serie de preguntas para establecer el archivo de configuración de este despliegue. En este caso, las respuestas a las preguntas fueron las siguientes:

```
Stack Name [sam-app]: serverless-app
AWS Region [us-east-1]:
#Shows you resources changes to be deployed and require a 'Y' to initiate deploy
Confirm changes before deploy [Y/n]:
#SAM needs permission to be able to create roles to connect to the resources in
your template
Allow SAM CLI IAM role creation [Y/n]: n
Capabilities [['CAPABILITY_IAM']]:
#Preserves the state of previously provisioned resources when an operation fails
Disable rollback [y/N]:
addSuplierFunction has no authentication. Is this okay? [y/N]: y
deleteSupplierFunction has no authentication. Is this okay? [y/N]: y
getAllSuppliersFunction has no authentication. Is this okay? [y/N]: y
getSupplierByIdFunction has no authentication. Is this okay? [y/N]: y
updateSupplierFunction has no authentication. Is this okay? [y/N]: y
Save arguments to configuration file [Y/n]:
SAM configuration file [samconfig.toml]:
SAM configuration environment [default]:
```

Esto genera un changeset, el cual es necesario confirmar para que inicie la creación, en un stack de CloudFormation, de los recursos definidos en el template.

Cuando CloudFormation termina de crear el stack, la infraestructura para la aplicación está desplegada. Sin embargo, para poder acceder a la aplicación web hay que subir manualmente los archivos de la web al bucket creado por el stack de CloudFormation.

5.4 Definición del Pipeline CI/CD

Para poder automatizar el despliegue del template anterior y el despliegue del frontend al bucket S3 cada vez que se realicen cambios en el repositorio de la aplicación es necesario la creación de un pipeline CI/CD, que será creado con el servicio de AWS CodePipeline junto con AWS CodeBuild.

Para la realización de esta tarea primero es necesario tener el código de la aplicación en un repositorio. En este caso, se usará un repositorio creado en AWS CodeCommit llamado `'serverless'`. Segundo, es necesaria la creación de otro template que defina los recursos necesarios para el pipeline.

AWS SAM ofrece una plantilla inicial para la creación de un pipeline a través del comando `'sam pipeline init –bootstrap'`. Esta plantilla inicial contiene la definición de recursos que son necesarios para la ejecución del pipeline, por ejemplo, los roles adecuados y un bucket S3 para guardar los artefactos que se generen en cada fase del pipeline y sean necesarios a la entrada de la siguiente fase. Se hará uso de dicha plantilla, aunque será modificada casi por completo para ajustarlo al caso de uso de este proyecto.

El pipeline tendrá 3 fases principales, como se indicó en el diagrama presentado en el Capítulo 4. La primera fase se denomina `'Source'`, en la que se obtiene el código del repositorio. La segunda fase se denomina `'BuildAndPackage'`, en la que mediante un proyecto de CodeBuild se ejecuta `'sam build'` para construir y compilar las funciones lambda, y demás artifactos necesarios, y `'sam package'` para empaquetar los artifactos y guardarlos en un bucket S3 para ser usados por la siguiente fase del pipeline. La tercera fase es la última de este pipeline, denominada `'Deploy'`. En esta fase se ejecuta `'sam deploy'` con los argumentos necesarios para desplegar el template de la arquitectura serverless. También se ejecuta un script para desplegar el frontend al bucket S3 que aloja el frontend.

Este pipeline contará con una fase extra para actualizar el propio recurso de pipeline. Esto quiere decir que, si se modifica el pipeline y se suben los cambios al repositorio, el pipeline iniciará y desplegará los cambios que se hayan hecho al mismo pipeline. Para entenderlo mejor, si se cambian las fases del pipeline y se agrega por ejemplo una fase de prueba, el pipeline después de la fase `'Source'` se modificará y agregará la nueva fase de prueba para ser usada inmediatamente después.

En la plantilla en la que se definirá el pipeline, también se definirá la creación de un bucket para el almacenamiento de los artefactos necesarios para desplegar la infraestructura serverless, además de dos roles para la ejecución adecuada del pipeline.

Una vez que se tiene la plantilla del pipeline definida, hay que subir todo al repositorio y desplegar la plantilla del pipeline con `'sam deploy -t codepipeline.yaml --capabilities=CAPABILITY_IAM --stack-name nombre_stack_pipeline'`

Cuando el stack se termine de desplegar, el pipeline se activará automáticamente y creara la infraestructura para la aplicación y el despliegue del frontend al bucket S3. Esto quiere decir que cuando el pipeline termine de ejecutarse, la aplicación quedará lista para ser accedida mediante la URL proporcionada por CloudFront.

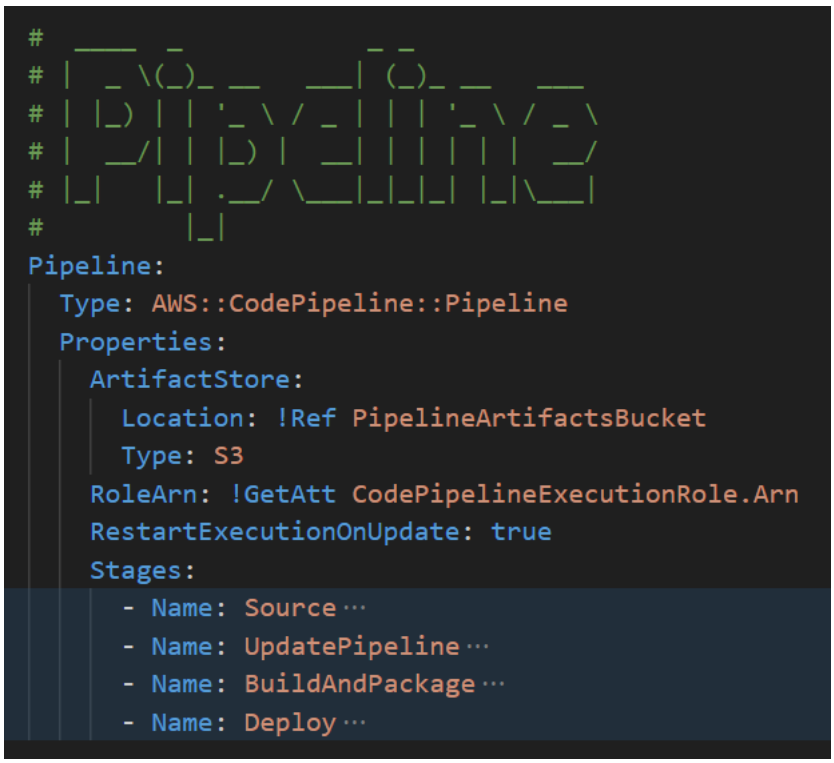

Figura 30. Definición del pipeline

6 Resultados y conclusiones

Con el desacoplamiento de la aplicación monolítica y su transformación en una aplicación con arquitectura serverless se han atacado efectivamente los puntos de problema principales mencionados en el análisis de la aplicación monolítica.

A continuación, se detallan y evalúan las ventajas que otorga el nuevo enfoque frente al antiguo.

- **Disponibilidad:** Debido a que los servicios usados en la construcción de la arquitectura serverless son completamente administrados por Amazon, son altamente disponibles. En contraste, una instancia EC2 por si sola no es altamente disponible debido a varias razones, entre las que se puede nombrar la ubicación en una sola zona de disponibilidad dentro de la región en la que se encuentre, posibles fallos de memoria, red o hardware que no puede manejar automáticamente y la posible necesidad de realizar mantenimientos de hardware o software por los cuales se interrumpe el servicio.

- **Escalabilidad:** Si existen picos de trabajo muy altos, la nueva arquitectura no tiene ningún problema en escalar automáticamente y manejar la alta demanda, gracias nuevamente al uso de servicios serverless. Por otro lado, la arquitectura antigua no sería capaz ni si quiera de escalar por si sola, lo que podría ocasionar cuellos de botella y perjudicar sustancialmente al rendimiento.

- **Rendimiento:** Un aspecto de mejora también es el rendimiento. Es evidente que gracias a la escalabilidad que tiene la nueva arquitectura, el rendimiento mejora notablemente. Pero, además, gracias a CloudFront la distribución de la página web se hace desde los servidores más cercanos al cliente. Esto junto con la caché que proporciona CloudFront hace que incluso en los momentos con más alta demanda, la nueva aplicación pueda tener un buen rendimiento y tiempos de respuesta muy bajos.

- **Seguridad:** Todos los datos almacenados en DynamoDB y buckets S3 son encriptados automáticamente en reposo. Además, CloudFront tiene una protección integrada contra ataques de denegación de servicio (DDoS) que la aplicación monolítica original no tiene.

- **Productividad y nuevas funcionalidades:** Gracias a la creación del pipeline, solo es necesario hacer los cambios que consideres oportunos en el código y subirlo a tu repositorio. Inmediatamente, el pipeline se activa y empieza el proceso de despliegue. Es importante mencionar que en el proceso de despliegue solo se actualiza o crea lo cambios hechos en el repositorio, no la arquitectura entera. Esto permite enfocarte en tu lógica de negocio y no desperdiciar tiempo en el mantenimiento o creación de infraestructura. Además, gracias también a la arquitectura, se pueden implementar o probar nuevas funcionalidades fácil y

rápidamente gracias a poder replicar el stack con el que estás trabajando. Abre un mundo de posibilidades.

Para hacer una comparación de costos entre las dos arquitecturas, hay que plantear dos situaciones y asumir la suposición hecha en la estimación de costos.

La primera situación, es asumir que la cuenta de Amazon en la que se desplegará la arquitectura es nueva, lo cual permite beneficiarte de ciertos servicios de la capa gratuita de Amazon que solo tienen validez los 12 primeros meses.

Asumiendo esto, el costo aproximado de la arquitectura serverless junto con el pipeline, es de 0,84 USD durante los 12 primeros meses, como fue mencionado en la estimación de costo.

Por el otro lado, manteniendo la arquitectura antigua con todas las desventajas que implica, los 12 primeros meses tendrían un valor de 0 USD debido a la capa gratuita que se ofrece para Amazon RDS y EC2.

La segunda situación, es asumir que la cuenta de Amazon en la que se desplegará la arquitectura NO es nueva.

En este caso, los únicos servicios de la arquitectura serverless que tienen un límite de 12 meses en sus capas gratuitas, son Amazon S3 y API Gateway. Por lo cual, habría que sumar 4,56 USD en concepto de API Gateway, y otros 0,08 USD en concepto de API Gateway. Como resultado final, los 12 primeros meses cuestan 5,47 USD.

En cambio, para la arquitectura antigua hay que sumar 132,72 USD en concepto de Amazon RDS, y 72,72 en concepto de EC2. Como resultado final, los 12 primeros meses cuestan 205,44 USD.

Es una diferencia abismal. Como conclusión, pienso que es mucho mejor migrar hacia la arquitectura serverless aunque el primer año con cuentas nuevas cueste 0,84 USD más caro que la arquitectura monolítica.

La nueva aplicación tiene varias posibilidades de mejora, como usar Amazon Cognito para crear un login en la página web o control de acceso a la API mediante Cognito. Otra posible mejora es agregar a la arquitectura AWS WAF, lo cual mejoraría aún más la seguridad gracias a la posibilidad de creación de reglas que controlen el tráfico de bots o los ataques más comunes.

7 Análisis de Impacto

7.1 Impacto Personal

La realización de este proyecto me ha ayudado a profundizar mis conocimientos sobre AWS en general, pero más específicamente en aquellos servicios que forman parte de las arquitecturas serverless. Entre estos servicios están AWS Lambda, Amazon CloudFront, DynamoDB y API Gateway. Además, he tenido la oportunidad de aprender cómo se automatiza la creación de infraestructuras y lo beneficioso que puede llegar a ser. Aprendí también más sobre CI/CD, y descubrí los beneficios prácticos que te ofrece el despliegue automatizado. Por último, aprendí a diseñar y realizar una estimación de coste de una arquitectura.

7.2 Impacto Empresarial

Como se mencionó anteriormente en las conclusiones, a nivel de empresa el impacto es muy grande. Esto es porque obtienes escalabilidad de tu producto, lo cual mejora el rendimiento, que, a su vez, mejora la experiencia de usuario. También multiplica tu productividad al no tener que encargarte de mantenimiento ni de la creación manual de infraestructura. Lo cual permite que la empresa se centré en su lógica de negocio.

7.3 Impacto Social y Económico

La migración hacia arquitecturas serverless puede permitir la creación de oportunidades laborales en otras empresas, que a su vez pueden ofrecer formación en el campo de computación en la nube.

7.4 Impacto Medioambiental

Los servicios serverless están diseñados y optimizados para ser altamente eficientes en el consumo de recursos. Debido a su escalabilidad en momentos de mucho tráfico, evitan el desperdicio de recursos con lo cual se disminuye la huella de carbono.

7.5 Relación con los Objetivos de Desarrollo Sostenible

El proyecto se relaciona con varios objetivos de Desarrollo Sostenible.

- ODS 7 – Energía Asequible y No Contaminante: Amazon se ha comprometido a operar sus centros con energía 100% renovable.

- ODS 12 – Producción y Consumo Responsables: Los servicios serverless optimizan los recursos computacionales y reducen la necesidad de infraestructura física, lo cual ayuda a una producción más sostenible y responsable.

8 Bibliografía

[1] Red Hat, "What is CI/CD?," Red Hat, 2023. [Online]. Available: https://www.redhat.com/es/topics/devops/what-is-ci-cd. [Accessed: May, 2024].

[2] Amazon Web Services, "CI/CD for 5G Networks on AWS," AWS, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/cicd_for_5g_networks_on_aws/cicd-on-aws.html. [Accessed: May, 2024].

[3] Amazon Web Services, "AWS CodeCommit," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codecommit/. [Accessed: May, 2024].

[4] Amazon Web Services, "AWS CodeBuild," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codebuild/. [Accessed: May, 2024].

[5] Amazon Web Services, "AWS CodePipeline," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codepipeline/. [Accessed: May, 2024].

[6] Amazon Web Services, "AWS CodeDeploy," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codedeploy/. [Accessed: May, 2024].

[7] Amazon Web Services, "AWS CloudFormation," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/cloudformation/. [Accessed: May, 2024].

[8] Amazon Web Services, "Arquitecturas serverless," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/lambda/serverless-architectures-learn-more/. [Accessed: May, 2024].

[9] Amazon Web Services, "Decomposing monoliths into microservices," AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/welcome.html. [Accessed: May, 2024].

[10] Wikipedia, "Domain-driven design," Wikipedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Domain-driven_design. [Accessed: May, 2024].

[11] Red Hat, "What is Blue-Green Deployment?," Red Hat, 2024. [Online]. Available: https://www.redhat.com/es/topics/devops/what-is-blue-green-deployment. [Accessed: May 30, 2024].

9 Anexos

TFG.pdf

by SANTIAGO DARIO MONTERO CABEZAS

Submission date: 03-Jun-2024 07:20PM (UTC+0200)
Submission ID: 2394751289
File name: 17766_SANTIAGO_DARIO_MONTERO_CABEZAS_TFG_530678_1688302895.pdf (2.58M)
Word count: 10547
Character count: 58566

Universidad Politécnica
de Madrid

**Escuela Técnica Superior de
Ingenieros Informáticos**

Grado en Ingeniería Informática

Trabajo Fin de Grado

Diseño y Despliegue de una Solución CI/CD y Serverless con AWS

Autor: Santiago Dario Montero Cabezas

Tutor(a): Sonia Frutos Cid

Madrid, mayo 2024

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

Trabajo Fin de Grado
Grado en Ingeniería Informática
Título: Diseño y Despliegue de una Solución CI/CD y Serverless con AWS
Mayo 2024

Autor: Santiago Dario Montero Cabezas

Tutor:
Sonia Frutos Cid
Departamento de Lenguajes y Sistemas Informáticos e Ingeniería de Software
ETSI Informáticos
Universidad Politécnica de Madrid

Resumen

En el trabajo presentado a continuación se realiza el desacoplamiento de una aplicación monolítica hacia una arquitectura serverless con una solución CI/CD integrada.

Para la realización de este proyecto se usan los servicios proveídos por Amazon Web Services, mediante los cuales se construirá el proyecto entero. Se realiza una selección de servicios a usar, el diseño de una arquitectura y la estimación de costo para la arquitectura diseñada.

Entre los servicios usados para la arquitectura se encuentran AWS Lambda, API Gateway, AWS CloudFront, Amazon S3 y DynamoDB. La automatización de la creación de la infraestructura se realiza usando una plantilla de AWS SAM.

Una vez definida la plantilla de la arquitectura se construye un pipeline para automatizar el despliegue de la infraestructura cuando se produzcan cambios en un repositorio. Este también se realiza a través de una plantilla de AWS SAM. Los servicios usados para la construcción del pipeline CI/CD se encuentran AWS CodePipeline, AWS CodeCommit y AWS CodeBuild.

Abstract

In the presented work, the decoupling of a monolithic application into a serverless architecture with an integrated CI/CD solution is carried out.

For the realization of this project, the services provided by Amazon Web Services are used, through which the entire project will be built. A selection of services to be used is made, the design of an architecture is created, and the cost estimation for the designed architecture is performed.

Among the services used for the architecture are AWS Lambda, API Gateway, AWS CloudFront, Amazon S3, and DynamoDB. The automation of the infrastructure creation is done using an AWS SAM template.

Once the architecture template is defined, a pipeline is built to automate the deployment of the infrastructure when changes are made in a repository. This is also done through an AWS SAM template. The services used for building the CI/CD pipeline include AWS CodePipeline, AWS CodeCommit, and AWS CodeBuild.

Tabla de contenidos

1 Introducción

1.1 Contexto y Motivación del Proyecto

En los últimos años, debido a la transformación digital, se ha aumentado el uso y la adopción de tecnologías cloud alrededor del mundo debido a sus beneficios. Entre sus beneficios podemos encontrar la escalabilidad, flexibilidad y reducción de costes operativos.

A pesar de esto, podemos encontrar arquitecturas tradicionales basadas en aplicaciones monolíticas que presentan serias limitaciones ya que suelen ser rígidas, costosas de mantener y presentan también problemas a la hora de escalar frente a picos de alta demanda por parte de los usuarios. Así, las arquitecturas serverless surgen como una solución revolucionaria debido a su escalabilidad automática, menor carga de mantenimiento y aún más optimización de costes operativos. Las arquitecturas serverless se basan en la implementación de funciones que se ejecutan en respuesta a eventos, como solicitudes HTTP, y solo pagas por el tiempo de ejecución y los recursos usados, como la memoria y el almacenamiento.

Además, es importante mencionar la importancia de prácticas DevOps hoy en día. La implementación de pipelines de integración y desarrollo continuo (CI/CD) facilitan la automatización de procesos de desarrollo y despliegue de software. Amazon Web Services (AWS), nos ofrece una gran variedad de servicios y herramientas que nos permiten la implementación de soluciones serverless y CI/CD.

Es en este contexto en donde se sitúa este Trabajo de Fin de Grado, que tiene como fin el diseño y despliegue de una solución CI/CD y serverless con AWS. La motivación principal de este proyecto tiene que ver con la necesidad de la transformación de una arquitectura monolitica hacia una arquitectura serverless para superar sus limitaciones, abrir las puertas a futuras innovaciones, y además permitir la automatización de la creación de la infraestructura (IaC) que ayude a los desarrolladores a centrarse en la lógica de negocio.

1.2 Objetivos Generales y Específicos

El objetivo de este trabajo es el diseño y despliegue de una arquitectura serverless en AWS, transformando una aplicación web monolítica desarrollada en Node.js conectada a una base de datos relacional en una arquitectura serverless.

Para conseguir este objetivo habrá que:

- Estudiar los servicios cloud AWS de CI/CD.
- Desacoplar una aplicación monolítica en funciones escalables.
- Migrar la base de datos a un servicio serverless.
- Definir un pipeline CI/CD.
- Automatizar de la infraestructura y creación de servicios cloud.
- Estimar costes.

1.3 Estructura del documento

- **Capítulo 1: Introducción –** Presenta el contexto y la motivación del proyecto, los objetivos generales y específicos, y la estructura del documento.

- **Capítulo 2: Estado del arte –** Examina los servicios cloud de AWS para CI/CD, las arquitecturas serverless, el desacoplamiento de las aplicaciones monolíticas y las técnicas de despliegue blue/green.

- **Capítulo 3: Metodología –** Describe el enfoque general del proyecto, las herramientas y tecnologías utilizadas, y la planificación del trabajo.

- **Capítulo 4: Diseño de la Arquitectura -** Analiza la aplicación monolítica existente, diseña la nueva arquitectura serverless, selecciona los servicios AWS adecuados y presenta el esquema de la infraestructura.

- **Capítulo 5: Desarrollo –** Detalla la creación del entorno de desarrollo, el desacoplamiento de la aplicación monolítica, la automatización de la creación de la infraestructura y la definición del pipeline CI/CD.

- **Capítulo 6: Resultados y Conclusiones –** Presenta los resultados y conclusiones, y compara los costes entre la aplicación monolítica y la aplicación serverless.

- **Capítulo 7: Análisis de Impacto -** Analiza el impacto del proyecto a nivel personal, empresarial, social, económico, medioambiental, y su relación con los Objetivos de Desarrollo Sostenible (ODS) de la Agenda 2030.

2 Estado del arte

En este capítulo se revisarán los conceptos y tecnologías que son importantes para el proyecto.

2.1 Servicios Cloud de AWS para CI/CD

Primero hay que entender que significa CI/CD. Como fue mencionado antes, CI/CD se refiera a la integración y desarrollo continuo cuyo objetivo es mejorar y agilizar el ciclo de vida de desarrollo de software [1]. Ambas forman parte de las practicas DevOps. Están diseñadas para mejorar la calidad del software y acelerar el despliegue a producción, esto también beneficia a disminuir el riesgo de los potenciales errores al momento de desplegar manualmente una aplicación.

La integración continua (CI) es una práctica donde se integra el código continuamente en un repositorio compartido. Esta integración construye y testea el código para asegurar que los cambios que se hagan sean fiables y no causen fallos en el código previo.

Por otro lado, la distribución continua (CD) se refiere a la automatización del despliegue de la aplicación una vez que esta se haya construido y haya pasado las pruebas correspondientes.

Según AWS, el CI/CD puede ser visto como una "tubería" en donde el nuevo código es introducido, por el inicio de la tubería, para pasar por un proceso de construcción, testeo y despliegue, para finalmente entregar código listo para producción [2].

AWS ofrece una gran variedad de herramientas y servicios para la creación de pipelines CI/CD.

2.1.1 AWS CodeCommit

Este es un servicio que te permite alojar repositorios de Git privados, administrados totalmente por AWS y con una alta escalabilidad. [3]

Entre otras funcionalidades, AWS CodeCommit te permite:

- Transferir archivos con HTTPS o SSH hacia o desde el repositorio.
- Almacenar cifrada y redundantemente el repositorio para aumentar la disponibilidad.
- Participar fácilmente en desarrollo de software colaborativo.

Para crear un repositorio se puede usar la consola de Amazon, los SDK o el interfaz de línea de Amazon (AWS CLI).

2.1.2 AWS CodeBuild

Este es un servicio de integración continua que te permite compilar código fuente, testear el código y producir software empaquetado listo para ser desplegado [4]. Es un servicio completamente administrado, lo que quiere decir que no tienes que preocuparte por crear tus servidores de compilación y testeo, ya que AWS te lo proporciona, con la opción de montar entornos de compilación ya creados o proporcionar el tuyo. Además de todo esto, AWS CodeBuild escala automáticamente para que no tengas que preocuparte por colas innecesarias.

2.1.3 AWS CodePipeline

Este servicio es el que te permite definir tu pipeline, es decir, con AWS CodePipeline se define las etapas del proceso de lanzamiento de software [5]. Una vez más, es un servicio totalmente administrado. Lo que nuevamente permite dejar a un lado la creación de nuestros propios servidores.

2.1.4 AWS CodeDeploy

Este servicio administrado, automatiza los despliegues de software en diferentes entornos, como desarrollo, pruebas y producción. Permite el despliegue en distintos servicios propios de AWS, como AWS Lambda, Amazon Elastic Compute Cloud (EC2) o Amazon Elastic Container Service (ECS) [6]. Por último, permite deshacer los despliegues si es necesario.

2.1.5 AWS CloudFormation

Es un servicio con el que se puede implementar y seguir tu infraestructura de AWS controlada y precisamente. Permite el modelado y diseño de tus recursos AWS dentro de tu infraestructura a través de ficheros de texto con formato JSON o YAML [7]. Esto facilita la creación de la infraestructura, y, además, permite replicar fácilmente dicha infraestructura para crear diferentes entornos, como entornos de prueba o producción.

Una de las capacidades más poderosas de CloudFormation es la capacidad de gestionar dependencias entre recursos de manera sofisticada. Puede modelar y gestionar las relaciones y dependencias entre los recursos, de tal manera que los recursos se desplieguen secuencialmente y se mantengan en un estado coherente. Esto beneficia al trabajar con infraestructuras que contienen un número considerable de recursos y servicios. Por último, CloudFormation minimiza la posibilidad de errores y el tiempo de inactividad.

2.2 Arquitecturas Serverless

Las arquitecturas serverless permiten a los desarrolladores centrarse en el código en lugar de en el mantenimiento del servidor. En lugar de gestionar la infraestructura y la lógica del servidor, simplemente se despliega una función que se ejecuta solo cuando se dispara un evento especifico, por ejemplo, una modificación en una base de datos o una solicitud HTTP entrante.

AWS Lambda es el servicio que da vida a esta forma de arquitectura, es capaz de ejecutar código sin tener que preocuparte por la administración de la maquina servidora. Con Lambda, lo único por lo que realmente pagas es el tiempo de ejecución, permitiendo así, optimizar los costos.

Existen también otros servicios que complementan Lambda y que ayudan a construir una arquitectura serverless completa. Entre esos otros servicios, tenemos a ApiGateway que te permite la creación y gestión de APIs, que pueden servir como eventos para lanzar las funciones Lambda. También existe AWS Step Functions, que permite la orquestación de funciones Lambda para crear flujos de trabajo tan grandes y complejos como quieras.

Las arquitecturas serverless tienen varios casos de uso debido a su escalabilidad y flexibilidad. Uno de los casos más comunes es el procesamiento de eventos en tiempo real, donde se utiliza AWS Lambda para el procesamiento en tiempo real de eventos emitidos por servicios como Amazon Kinesis o Amazon DynamoDB Streams [8]. Esto permite reaccionar automáticamente a eventos que pueden considerarse críticos, como transacciones financieras, cambios de inventario o análisis de las redes sociales. Por último, otro caso común es el de aplicaciones web o solo backend de aplicaciones.

2.3 Desacoplamiento de Aplicaciones Monolíticas

2.3.1 Definición de Aplicaciones Monolíticas

Es necesario empezar definiendo que es una aplicación monolítica. Una aplicación monolítica es un tipo de aplicación en el que todos sus componentes funcionan y se despliegan juntos como una sola unidad.

Las aplicaciones monolíticas son caracterizadas por lo siguiente:

- **Módulos fuertemente acoplados:** Todos los módulos y componentes están fuertemente acoplados. Existe una fuerte dependencia entre módulos que puede imposibilitar el desarrollo de funcionalidades nuevas o la corrección de errores sin tener que cambiar el sistema entero.

- **Escalabilidad limitada:** La manera de escalar estas aplicaciones son verticalmente, esto es, agregar más recursos de procesamiento en los servidores de las aplicaciones. Esto lo hace menos viable debido a sus costos y capacidad con respecto al escalado en horizontal.

- **Mantenimiento y evolución compleja:** Con el paso del tiempo, estas aplicaciones se vuelven difíciles de mantener. Además, para agregar nuevas funcionalidades es requerido que los programadores conozcan bien el programa entero debido a su fuerte acoplamiento, lo que puede disminuir la productividad y la fiabilidad del programa por el riesgo de introducir errores.

- **Eficiencia y simplicidad inicial:** En los primeros días de desarrollo, se puede implementar y poner en funcionamiento una aplicación monolítica con bastante facilidad y en poco tiempo. Sin embargo, con el crecimiento de la aplicación y el paso del tiempo, toda la simplicidad inicial se pierde.

2.3.2 Estrategias para Desacoplar Aplicaciones Monolíticas

Antes de empezar, es importante evaluar que aplicaciones monolíticas deberían descomponerse. Hay que descomponer monolitos que tengan problemas de fiabilidad o rendimiento, o que tengan una arquitectura fuertemente acoplada. Es importante también conocer la aplicación, conocer las tecnologías que usa y cuál es su uso u objetivo de negocio [9].

A continuación, se presentan los patrones usados.

2.3.2.1 Descomposición por departamentos o áreas

Es posible usar la organización por departamentos o áreas para descomponer los monolitos en microservicios. Generalmente las organizaciones se dividen en departamentos, por ejemplo, departamento de ventas, departamento de marketing o departamento de atención al cliente [9].

- **Ventajas:** Los equipos de trabajo estarán bien organizados por el departamento y microservicio al que pertenezcan. Esto genera una arquitectura de microservicios estable en el que se encuentran débilmente acoplados [9].
- **Desventajas:** Se requiere un conocimiento acerca de la organización para entender sus divisiones y, además, el diseño de la arquitectura estaría fuertemente acoplado con su división departamental [9].

2.3.2.2 Descomposición por subdominio

Este patrón aprovecha el diseño basado en dominios [10] para descomponer los monolitos según subdominios. Lo que se hace es aprovechar el límite bien definido que existe entre los módulos relacionados con los subdominios para empaquetar dichos módulos en nuevos microservicios [9].

- **Ventajas:** Se vuelven sistemas más escalables y que tienen una arquitectura débilmente acoplada [9].
- **Desventajas:** Se necesita un buen conocimiento acerca de la organización para poder identificar los subdominios. Por otro lado, puede generar demasiados microservicios, volviendo difícil integrarlos [9].

2.3.2.3 Descomposición por transacciones u operaciones

Este patrón descompone los monolitos según la operación o transacción que se deba realizar, esto quiere decir que, las aplicaciones suelen tener que llamar a varios microservicios para realizar una operación o transacción, por lo cual, agrupando los microservicios necesarios para una transacción se puede reducir la latencia y el número de llamadas necesarias. Es adecuado si al agrupar dichos microservicios no formas un monolito [9].

- **Ventajas:** Mejora sobre la consistencia de los datos, los tiempos de respuesta y la disponibilidad [9].
- **Desventajas:** Existe un riesgo de agrupar microservicios que formen un monolito, así como de aumentar la complejidad del microservicio [9].

2.3.2.4 Servicio por equipo de trabajo

Este patrón, como su nombre lo indica, descompone los monolitos en microservicios por equipos de trabajo en lugar de por departamentos, es decir, asigna se crea un microservicio que será totalmente gestionado por cada equipo de trabajo [9].

- **Ventajas:** Cada equipo puede gestionar su microservicio con la tecnología que prefiera, actuando independientemente. Permite un rápido avance e innovación en las funcionalidades del producto [9].
- **Desventajas:** Puede resultar difícil cuando existen dependencias circulares entre equipos [9].

2.3.2.5 Patrón "Strangler fig"

Este patrón envuelve la aplicación monolítica entera en un nuevo sistema, el cual se va incrementalmente transformando en microservicios. Está pensado para sistemas grandes y antiguos, que quieran ser modernizados y permita la coexistencia del sistema antiguo con los nuevos microservicios modernos que se van implementando [9].

- **Ventajas:** Permite agregar nuevos servicios mientras se siguen refactorizando los antiguos, así como también permite la interacción con servicios antiguos que no son ni serán actualizados. Por último, tiene la capacidad de mantener el servicio antiguo en línea mientras se refactoriza el código para actualizarlo [9].
- **Desventajas:** No es posible usarlo en sistemas donde no se puedan interceptar las solicitudes al backend, así como tampoco debería usarse en sistemas pequeños con baja complejidad [9].

2.3.2.6 Patrón de ramificación por abstracción

Este patrón se usa cuando no es posible interceptar las llamadas hacia el monolito y quieres modernizar componentes muy arraigados profundamente en el sistema, con dependencias fuertes [9]. En el siguiente ejemplo se puede entender mejor su funcionamiento.

Imagina que tienes un módulo de procesamientos de pagos que depende de otros componentes, facturación y gestión de pedidos. Necesitas crear una interfaz que defina las interacciones entre el módulo de pagos y sus módulos clientes, facturación y gestión de pedidos. Luego, modificas el módulo de facturación y gestión de pedidos para que usen la interfaz en lugar del módulo de pagos directamente. Por último, desarrollas la nueva implementación del módulo de pagos, con tecnologías modernas y escalables, y cambias la interfaz por el nuevo módulo de pagos.

- **Ventajas:** Permite la coexistencia de múltiples implementaciones de un módulo de software, cambios incrementales reversibles y modernizar funcionalidad que está muy dentro del sistema y a la cual no puedes interceptar sus llamadas [9].
- **Desventajas:** El esfuerzo puede no valer la pena debido a tener que reestructurar demasiado, y a sistemas pobremente estructurados [9].

2.4 Despliegue Blue/Green

Cuando se habla de despliegues blue/green, se habla de un modelo de lanzamiento en el que el tráfico se va redirigiendo gradualmente de una aplicación o microservicio en producción hacia una nueva versión de esta.

La versión blue, o azul, es la versión actual o previa. En cambio, la versión green, o verde, es la versión nueva, a la cual será redirigida todo el tráfico. Cuando todo el tráfico ha sido completamente redirigido, la versión blue se puede conservar en caso de que sea necesario volver a la versión anterior [11].

El uso de implementaciones blue/green traen consigo una serie de ventajas:

- Disminución del tiempo de inactividad.
- Disminución del riesgo de errores.
- Posibilidad de revertir la versión fácilmente en caso de problemas.
- Facilidad para la realización de pruebas.

De estas ventajas se benefician específicamente las aplicaciones críticas, donde el tiempo de inactividad deber ser mínimo. Además, también se benefician los entornos donde las actualizaciones de software son frecuentes y deben ser gestionadas sin grandes interrupciones.

3 Metodología

3.1 Enfoque General del Proyecto

Se ha optado por seguir un enfoque ágil para llevar a cabo este proyecto. Este tipo de metodología permite una gestión de proyectos flexible y reactiva que se ajusta según sea necesario. Esto permite amoldarse a posibles cambios o problemas que puedan surgir. Con este tipo de proyectos donde es necesario integrar varias tecnologías y servicios, la metodología ágil funciona muy bien.

3.2 Herramientas y Tecnologías Utilizadas

La aplicación web facilitada por Amazon que será desplegada como una infraestructura serverless, usa Node.js junto con Express y MustacheExpress. Para su transformación a estructura serverless, se ha decidido usar Vue.js para el frontend de la aplicación. Esta decisión ha sido tomada debido a que Vue.js permite el renderizado en el lado del cliente, facilitando así el despliegue en algún servicio relacionado con infraestructuras serverless.

Amazon Web Services (AWS), es sin duda, la herramienta principal y que da vida a este proyecto. La creación total de la infraestructura fue hecha con los servicios que AWS proporciona, y principalmente, el foco está puesto sobre AWS Lambda. AWS Lambda es uno de los principales servicios cuando hablamos de serverless. Más adelante, en el capítulo de Diseño de la Arquitectura, se expondrá con detalle cuales fueron los servicios específicos usados para construir la infraestructura en su totalidad.

Es importante mencionar que, al principio de este proyecto, mi conocimiento sobre AWS era equivalente al nivel de AWS Cloud Practitioner. Por lo cual, fue necesario adquirir conocimiento sobre los nuevos servicios necesarios para el desarrollo del proyecto.

Para poder trabajar con AWS existen varias maneras de hacerlo:

- **Consola de Administración de AWS:** Aplicación web proporcionada por Amazon que permite la gestión de los recursos AWS.
- **AWS CLI (Interfaz de Línea de Comandos):** Herramienta que permite interactuar con los servicios de AWS a través de comandos en tu Shell.
- **AWS SDKs:** Herramientas que permiten a los desarrolladores un acceso programatico a los servicios de AWS mediante APIs en cada lenguaje de programación especifico.
- **AWS CloudShell:** Shell preautenticado al que se puede acceder desde la consola de AWS y que permite interactuar con los servicios de AWS.

Para el desarrollo del proyecto se usó, en algún momento, cada una de las 4 posibilidades de interacción con AWS. Principalmente se utilizó la SDK de Python, AWS CLI y la consola de AWS.

Para poder hacer uso de AWS CLI es necesario configurar las credenciales de la cuenta AWS que usaremos, es decir, la cuenta AWS donde crearemos los recursos. Para la realización de este proyecto, se hace uso de una cuenta personal creada con este fin. Esto se debe a que, a pesar de que Amazon nos ofrece un "sandbox environment", tiene permisos muy limitados. Para configurar AWS CLI con este entorno ofrecido, primero es necesario realizar la instalación de este. Se usará AWS CLI para Windows, además de Visual Studio Code como centro de operaciones. Una vez instalado AWS CLI, se pueden configurar las credenciales mediante el comando *'aws configure'*.

Como se ha mencionado, se utiliza la herramienta VS Code junto con su extensión AWS Toolkit. Además, Git es esencial para el control de versiones.

Por último, se hace uso de AWS Serverless Application Model (AWS SAM), que es un conjunto de herramientas orientadas al desarrollo de aplicaciones serverless. AWS SAM te permite definir plantillas que luego pueden ser ejecutadas por AWS CloudFormation para la automatización de la creación de la infraestructura.

A continuación, se recopila la lista de herramientas y tecnologías utilizadas:

- Vue.js
- Amazon Web Services (AWS)
- Consola de AWS
- AWS CLI
- AWS SDK de Python
- Visual Studio Code
- Git
- AWS SAM

3.3 Planificación del Trabajo

3.3.1 Lista de Tareas

Se ha definido una lista de tareas junto con su dedicación horaria para la realización del proyecto.

- Estudiar servicios cloud AWS de CI/CD (20h)
- Planificar el diseño de la arquitectura y estimar los costes (20h)
- Analizar la infraestructura de la aplicación monolítica (20h)
- Crear un entorno de desarrollo y un repositorio Git (2h)
- Desacoplar la aplicación monolítica en funciones (60h)
- Migrar la base de datos (2h)
- Automatizar la creación de infraestructura y servicios AWS (76h)
- Definir el pipeline CI/CD (50h)
- Documentación del proyecto (47h)

3.3.2 Diagrama de Gantt

A continuación, en la Figura 1, se puede apreciar el diagrama de Gantt final de este proyecto.

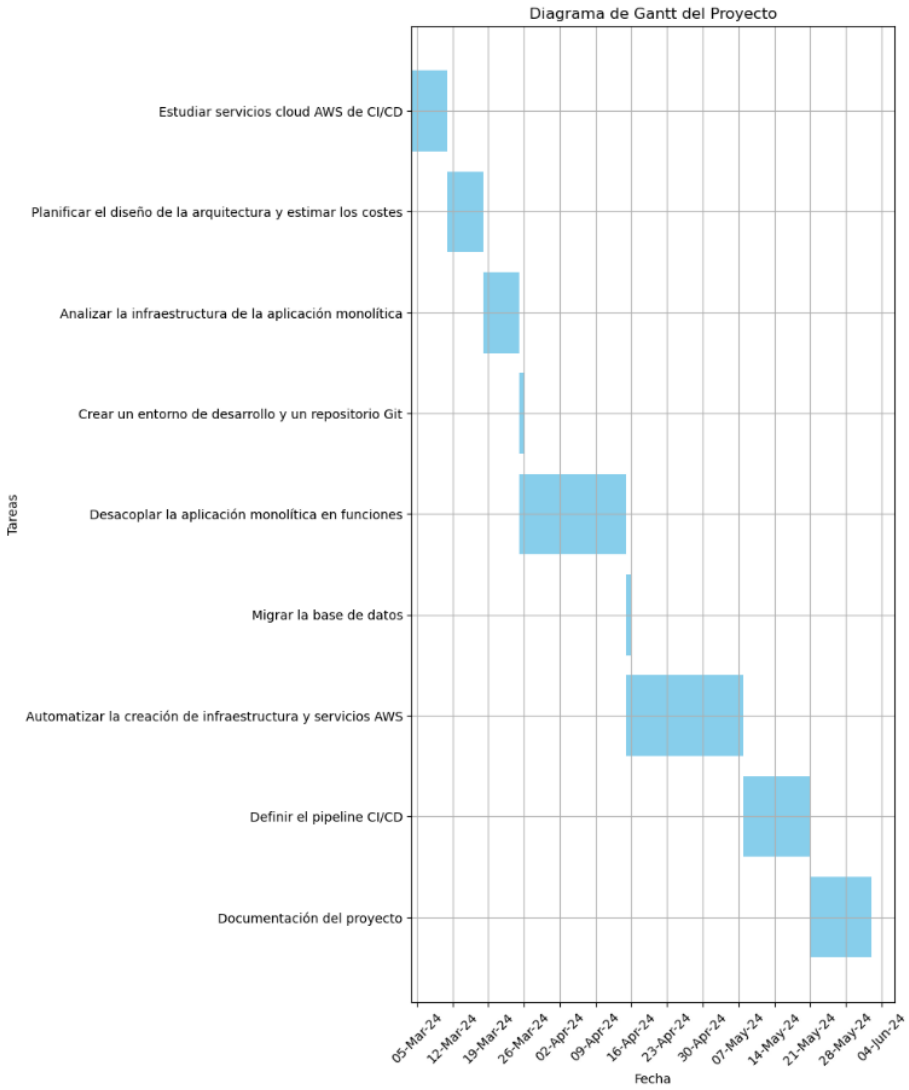

Figura 1. Gráfico de Gantt del Proyecto

4 Diseño de la Arquitectura

4.1 Análisis de la Aplicación Monolítica Existente

Para poder transformar la aplicación monolítica a una arquitectura serverless lo primero que hay que hacer es conocer la aplicación, entender sus funcionalidades y analizar cómo está estructurada, para luego poder diseñar la nueva arquitectura serverless.

La aplicación monolítica facilitada por Amazon para realizar este proyecto se encuentra desplegada en AWS, en un entorno de laboratorio para poder acceder a la aplicación y probar su funcionalidad. Además, allí se encuentra el código fuente de la aplicación.

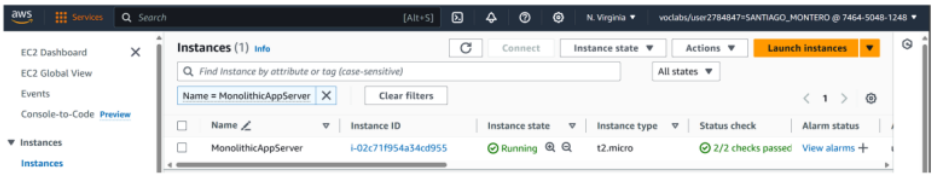

Figura 2. Consola AWS | EC2 Dashboard

Como se puede comprobar en la Figura 2, existe una instancia EC2 iniciada con el nombre de MonolithicAppServer.

En la Figura 3, se muestra el aspecto que tiene la aplicación web.

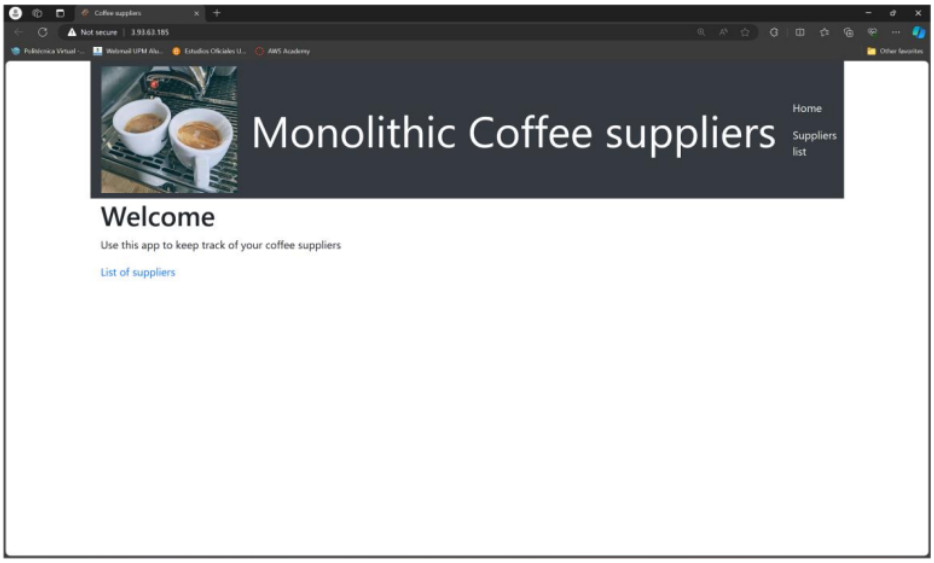

Figura 3. Aplicación web monolítica

La aplicación es bastante sencilla como se puede apreciar, y según el contexto ofrecido por Amazon, se trata de una aplicación que pertenece a una corporación de cafeterías con muchas franquicias que se está expandiendo constantemente y volviéndose muy popular. Es una aplicación de listado de proveedores que tiene problemas de rendimiento y fiabilidad.

La aplicación contiene la página de inicio, que se muestra en la Figura 3, y una página donde se lista a los proveedores junto con su información.

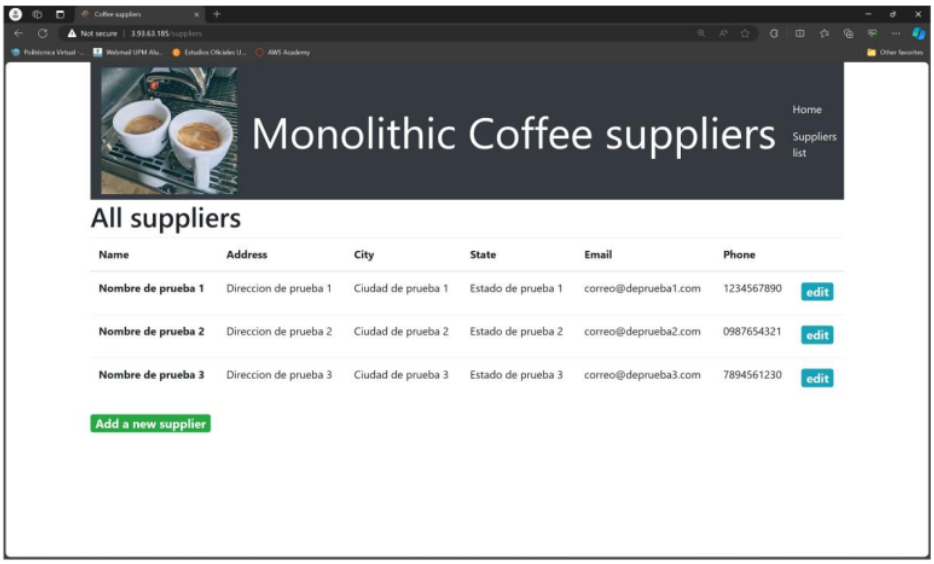

Figura 4. Página de listado de proveedores de la aplicación web

En la Figura 4, se aprecia la página en donde se listan los proveedores, donde además se puede observar que existen botones para agregar y editar la información de cada proveedor. Los botones nos llevan a un formulario que permiten también eliminar el proveedor

La aplicación, evidentemente, tiene persistencia. La instancia EC2 está conectada a una instancia de Amazon RDS, en donde guarda la información de los proveedores en una base de datos MySQL llamada COFFEE.

Esta base de datos solo contiene una tabla sin entradas llamada suppliers.

```
mysql> show full tables from COFFEE;
+-----------------+------------+
| Tables_in_COFFEE | Table_type |
+-----------------+------------+
| suppliers        | BASE TABLE |
+-----------------+------------+
1 row in set (0.00 sec)
```

Figura 5. Listado de tablas de la base de datos COFFEE

4.2 Diseño de la Nueva Arquitectura Serverless

Con el análisis hecho, podemos ver que la aplicación corre en una instancia EC2, es decir una máquina virtual, con 1GB de RAM y 1 vCPU (CPU virtual). Esto hace que se pueda generar un cuello de botella y que sea una solución no escalable, confirmando los problemas mencionados por Amazon: fiabilidad y rendimiento.

Se va a separar la aplicación en backend y frontend, en donde el frontend realice llamadas al backend a través de una API. Esta solución permite el uso de servicios serverless que formaran parte de la nueva arquitectura, y que se muestran a continuación.

4.2.1 Selección de Servicios AWS

A continuación, se listan los servicios AWS que se usaran para la nueva arquitectura, y también, los que forman parte de la solución completa del proyecto.

- **AWS Lambda:** Servicio para la implementación de las funciones que formaran el backend, por ejemplo, guardar proveedores, recuperarlos de la base de datos, etc.
- **API Gateway:** Servicio para la creación de una API que permita la comunicación bidireccional entre frontend y backend. También se encarga de redirigir las solicitudes a las correspondientes funciones lambda.
- **DynamoDB:** Servicio para la base de datos NoSQL serverless que funciona como almacenamiento de los datos de la aplicación de manera escalable y con alta disponibilidad.
- **Amazon S3:** Servicio que permite la creación de un bucket que almacenará los archivos estáticos necesarios para el frontend.
- **AWS CloudFront:** Servicio para la distribución de contenido web estático que tiene como origen el bucket de S3 con el frontend de la aplicación.
- **AWS CodeCommit:** Servicio para la creación del repositorio de la aplicación.
- **AWS CodeBuild:** Servicio para la construcción y despliegue automatizado de la aplicación.
- **AWS CodePipeline:** Servicio para la creación del pipeline CI/CD
- **AWS Identity and Access Management (IAM):** Servicio para la gestión de roles.
- **Amazon CloudWatch Events:** Servicio para la creación de reglas que funcionan como disparadores, en este caso, de la activación del pipeline.
- **AWS CloudFormation:** Servicio para la gestión de infraestructura como código (IaC), que permitirá ejecutar las plantillas de SAM para la creación de la arquitectura serverless y el pipeline.

4.2.2 Estimación de Costos

Para la estimación de costos es necesario hacer algunas suposiciones, ya que no tenemos datos reales de uso de la aplicación. Además, se tiene que considerar el crecimiento continuo de la aplicación y hacer una estimación de los posibles números de uso a los que la aplicación puede llegar.

Vamos a suponer que existen unas 300 franquicias, las cuales visitan la aplicación web varias veces al día. En concreto, vamos a suponer que cada franquicia realiza un promedio de 12 solicitudes (listado y manipulación) por día, y que para cada solicitud son necesarias en promedio 5 solicitudes HTTPS.

A continuación, se muestra la configuración de cada servicio para la estimación de costos.

- **AWS Lambda**
  - **Región:** US East (N. Virginia)
  - **Arquitectura:** x86
  - **Cantidad de solicitudes por mes:** 108.000
  - **Duración de cada solicitud (en ms):** 200
  - **Cantidad de memoria asignada:** 128 MB
  - **Cantidad de almacenamiento efímero asignado:** 512 MB
- **API Gateway**
  - **Región:** US East (N. Virginia)
  - **API de REST**
    - **Unidades de solicitud de la API REST:** miles
    - **Solicitudes por mes:** 108
- **DynamoDB**
  - **Región:** US East (N. Virginia)
  - **DynamoDB capacidad aprovisionada:** true
  - **Clase de tabla:** Estándar
  - **Tamaño del almacenamiento de datos:** 5 GB (Free Tier)
  - **Porcentaje de escrituras no transaccionales:** 100%
  - **Tasa de escritura de referencia:** 2 por segundo (Free Tier)
  - **Tasa de escritura máxima:** 2 por segundo (Free Tier)
  - **Porcentaje de lecturas altamente consistentes:** 100%
  - **Tasa de lectura de referencia:** 2 por segundo (Free Tier)
  - **Tasa de lectura máxima:** 2 por segundo (Free Tier)
- **Amazon S3**
  - **Región:** US East (N. Virginia)
  - **S3 Standard:** true
  - **Data transfer:** true
  - **Almacenamiento de S3 Estándar:** 1MB
  - **Solicitudes PUT a S3 por mes:** 10 (Free Tier)
  - **Solicitudes GET, SELECT por mes:** 20000 (Free Tier) + 61000
  - **Transferencia de datos a:** Amazon CloudFront (Free Tier)
- **Amazon CloudFront**
  - **Región:** US East (N. Virginia)
  - **Estados Unidos, Canadá, Europa:**
    - **Transferencia de datos salientes a Internet por mes:** 5,4 GB (Free Tier)
    - **Número de solicitudes HTTPS por mes:** 540000

- **AWS CodeCommit**
  - **Número de usuarios activos:** 3
- **AWS CodeBuild**
  - **Numero de compilaciones en un mes:** 10
  - **Duración media de la compilación (minutos):** 1
  - **Sistema operativo:** Linux
  - **Tipo de instancia de computación:** general1.small
- **AWS CodePipeline**
  - **Número de pipelines activos utilizados por cuenta al mes:** 1

Con las estimaciones iniciales, y unas pocas más que se hicieron a lo largo de la estimación por servicio, podemos dar una aproximación del costo durante los primeros 12 meses de la arquitectura desplegada en Amazon. Esto es solo una estimación, y no se incluyen los impuestos que puedan aplicar. El costo real podría variar por cuestiones impredecibles. La estimación aprovecha las ventajas de la capa Free Tier de Amazon, la cual te ofrece ciertos servicios gratis para toda la vida y otros solo durante los primeros 12 meses.

La estimación queda como se puede apreciar en la Figura 6 a continuación.

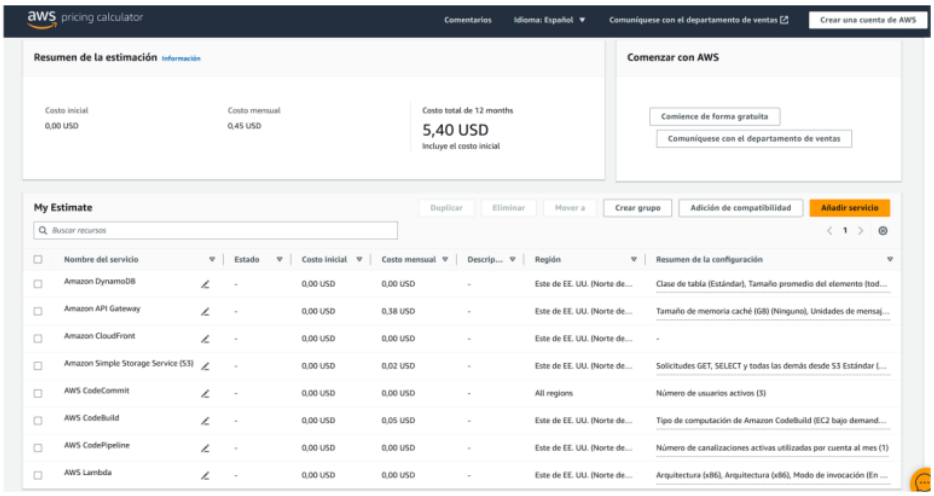

Figura 6. Resumen de la estimación realizada en AWS pricing calculator

Por lo cual, según nuestras suposiciones y la configuración mencionada, el costo de los primeros 12 meses de nuestra arquitectura serverless desplegada en Amazon es asombrosamente del valor de 5,40 USD, al que hay que restarle 4,56 USD gracias al Free Tier de API Gateway durante los primeros 12 meses. Quedando así el aún más asombroso valor de 0,84 USD.

4.2.3 Diagrama de la Arquitectura

Una vez que los servicios AWS que se utilizarán en la arquitectura serverless han sido escogidos, es momento de pasar a la construcción de esta. La nueva arquitectura se puede apreciar en el diagrama a continuación.

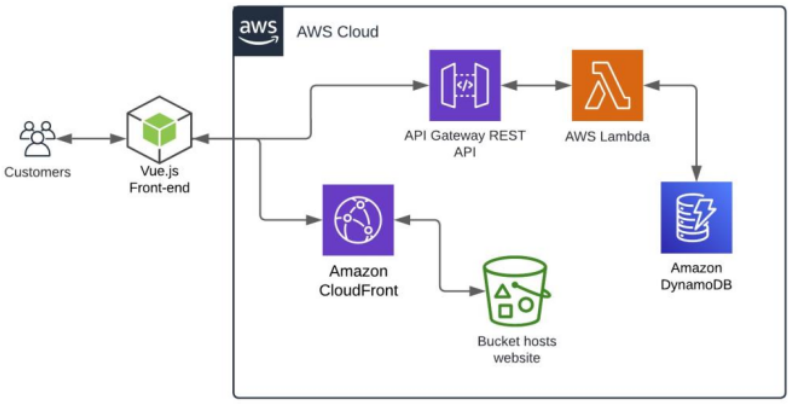

Figura 7. Diagrama de la Arquitectura Serverless

Los clientes (customers) interactuarán con el frontend desarrollado en Vue.js. Las peticiones HTTPS serán servidas por Amazon CloudFront, el cual tiene como origen un bucket de Amazon S3. Dicho bucket contendrá los recursos estáticos de la página web, como los HTML, CSS, Javascript e imágenes. Por otro lado, el backend de la aplicación estará conformado por una REST API. La cual contendrá los siguientes recursos.

- "url-api/suppliers" con métodos GET y POST.
- "url-api/suppliers/{id}" con métodos GET, PUT y DELETE

Por cada uno de los métodos y recursos habrá una función lambda conectada, la cual interactuará con una base de datos NoSQL cumpliendo con la solicitud enviada.

Amazon DynamoDB contendrá una tabla en la que almacenará toda la información de los proveedores. Como dato adicional, los accesos a DynamoDB tienen latencias muy bajas.

Cada servicio de la arquitectura es altamente disponible y cuenta con muy buena escalabilidad.

En la Figura 8, se puede ver el diagrama del pipeline. Los desarrolladores hacen cambios en el repositorio y el pipeline se activa. El código se construye y empaqueta, y después se despliegan los correspondientes cambios en los servicios AWS de la arquitectura serverless.

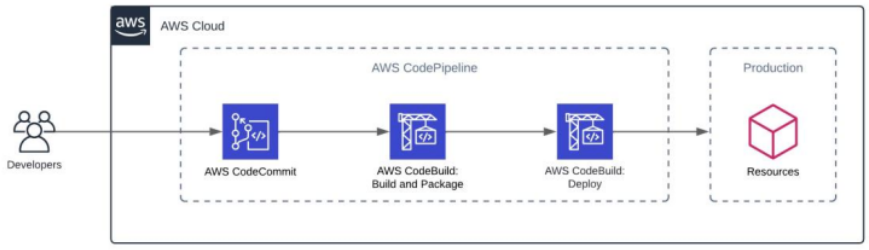

Figura 8. Diagrama del Pipeline

Con la nueva arquitectura diseñada, no solo se solucionan los problemas de escalabilidad, rendimiento y fiabilidad, sino que también se abre la puerta a futuros desarrollos de nuevas funcionalidades con una gran facilidad, ya que solo sería necesario agregar lo nuevo, sin tener que cambiar toda la aplicación. Además, el pipeline facilita el despliegue de la aplicación con su infraestructura, permitiendo productividad y eficiencia.

5 Desarrollo

5.1 Creación del Entorno de Desarrollo

Como se ha mencionado antes, para la realización de este proyecto fue necesario la creación de una cuenta en AWS.

Una vez con la cuenta creada, es hora de acceder a la cuenta y crear un usuario IAM con acceso programático mediante credenciales y permiso para el acceso a la consola de AWS. A este, se le adjunta el rol "AdministratorAccess" administrado por Amazon. Dicho rol permite la gestión completa de todos los servicios de AWS.

Con el usuario IAM creado junto con sus credenciales, se puede pasar a configurar AWS CLI para poder interactuar con AWS a través del comando `'aws configure'`. Las credenciales se tratan de un par de claves: AWS Access Key ID y AWS Secret Access Key. El comando es interactivo, en donde solo es necesario introducir las credenciales.

También es posible acceder a la consola de Amazon, mediante el usuario y contraseña que se definió a la hora de crear el usuario, para tener una interacción con AWS más gráfica.

Por último, hay que obtener el código de la aplicación monolítica. Como se mencionó anteriormente, este código es facilitado por Amazon y se encuentra la instancia EC2 que está desplegada en el entorno de laboratorio. El código tiene la estructura que se muestra en la Figura 9.

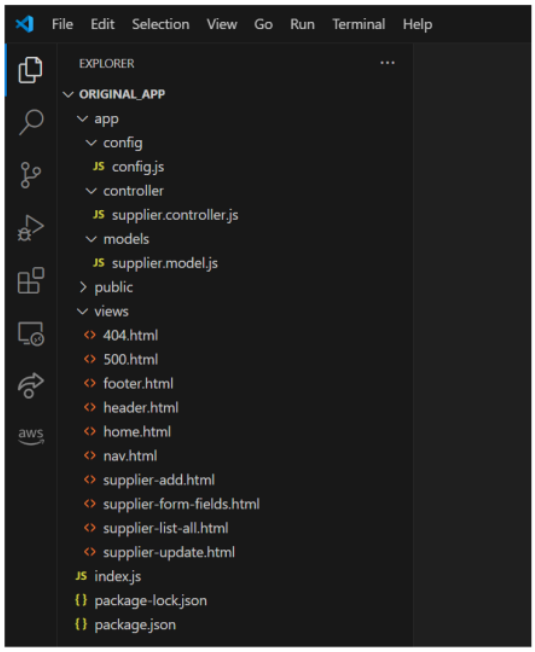

Figura 9. Estructura del código de la aplicación monolítica

5.2 Desacoplamiento de la Aplicación Monolítica

Para empezar con el desacoplamiento de la aplicación, hay que fijarse en cómo funciona el código actual. Para esto, hay que centrarse mayoritariamente en el punto de entrada principal de la aplicación, que es el archivo index.js.

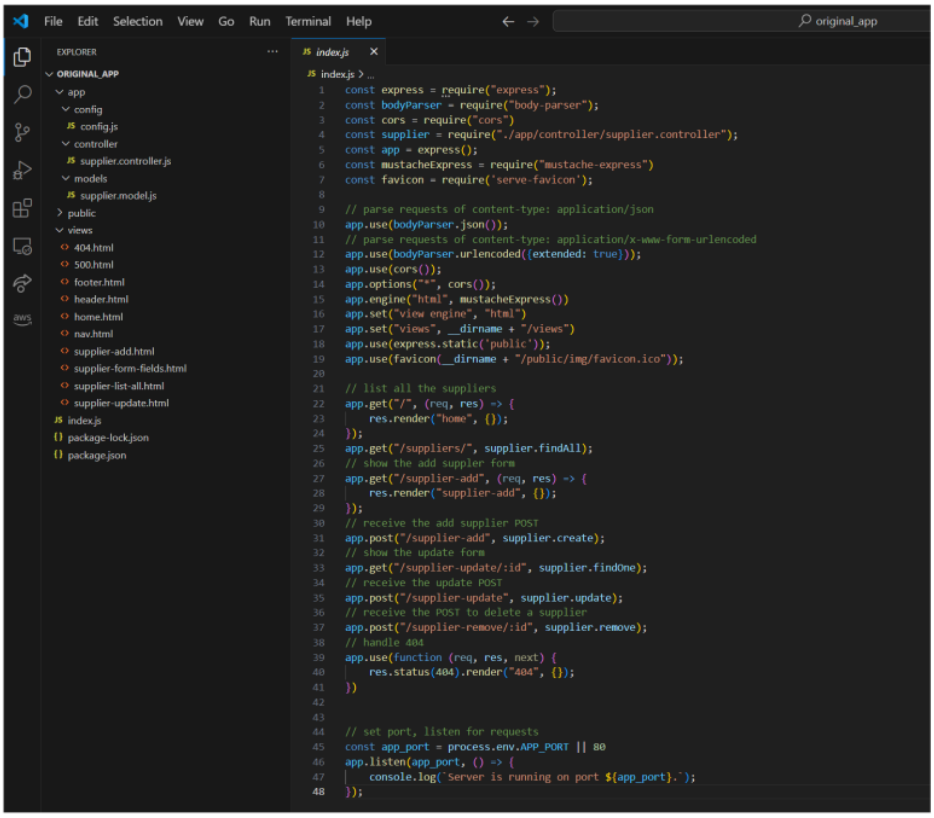

Figura 10. Punto de entrada principal a la aplicación

Con el código mostrado en la Figura 10 se puede extraer casi toda la información necesaria para entender el cómo funciona la aplicación.

Una de las cosas importantes es saber que usa Express.js para la creación de un servidor web, y que, además, usa mustacheExpress para la renderización de HTML.

Sin embargo, lo más importante es fijarse en las rutas que están definidas, las cuales sirven para agregar, listar, actualizar y eliminar los proveedores. Esto es lo más importante ya que aquí se marcará la separación entre el frontend y el backend, es decir, estas rutas serán los recursos de la API que se definirá como parte del backend que será usado por el frontend para realizar las llamadas respectivas.

El archivo config.js sirve para la crear la conexión con la base de datos, y el archivo supplier.model.js define el modelo de datos del proveedor y las funciones para interactuar con la base de datos. De este modo, estos archivos prácticamente no tienen utilidad, ya que para lo único que podían servir era para conocer el modelo de datos del proveedor, sin embargo, este se puede conocer gracias a la base de datos y la estructura de su tabla COFFEE vista anteriormente.

El archivo supplier.controller.js contiene la lógica del controlador para manejar las operaciones realizadas sobre los proveedores, y en qué desencadena cada solicitud. Esto se puede ver como la navegación que existe en la aplicación, lo cual servirá para construir el frontend.

Por último, se encuentra la carpeta de views que contiene los archivos HTML que definen las vistas de la aplicación, lo cual también será útil en la construcción del frontend.

Con este pequeño análisis realizado, se puede concluir que es necesaria la construcción de un frontend, reutilizando los archivos HTML y la lógica del controlador, que realice peticiones al backend a través de una API.

5.2.1 Frontend

Se usará Vue.js para la construcción del frontend debido a su renderización client-side, lo cual se acerca más a un enfoque serverless que tener un servidor web con Express corriendo en una máquina virtual que sería necesario administrar. Este frontend estará alojado en un bucket S3 y será distribuido por Amazon CloudFront.

¿Por qué no solo alojar el frontend en el bucket S3 como web estática? Esto se debe a que con Amazon CloudFront se pueden obtener ventajas significativas frente al uso de solo S3 como web estática. A continuación, se detallan las principales razones por las cuales se opta al uso de CloudFront:

- **Rendimiento:** CloudFront distribuye el contenido a través de sus "Edge Locations", lo que significa que las solicitudes se resuelven desde el servidor más cercano a los clientes. Esto sumado con el caching que realiza resultan en un mejor rendimiento con latencias y tiempos de carga bajos.
- **Seguridad:** CloudFront proporciona HTTPS sin la necesidad de configurar un certificado en S3. Además, tiene protección contra ataque DDoS integrada.
- **Flexibilidad y escalabilidad:** Debido a lo mencionado en el punto de rendimiento, CloudFront distribuye el tráfico uniformemente, lo cual mejora la escalabilidad en momentos de picos de tráfico.

Para construir esta aplicación en Vue.js se reutilizará los archivos contenidos en la carpeta "views" del código original de la aplicación.

Como resultado, en la Figura 11 se puede observar los componentes de Vue.js creados para el nuevo frontend.

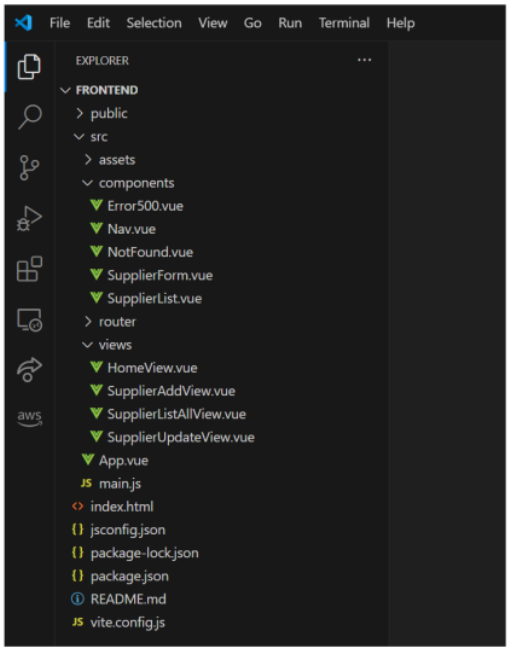

Figura 11. Estructura del código del nuevo frontend

Se crearon las 4 páginas principales que tenía la aplicación original, la página principal, la página donde se listan los proveedores, la página con el formulario para añadir un nuevo proveedor, y la página con el formulario para actualizar o eliminar un proveedor. Los demás componentes sirven para crear estas páginas, como por ejemplo SupplierForm.vue, que es el componente que contiene el formulario común para las páginas donde se muestra el formulario.

Una vez listo y probado localmente el frontend desarrollado con Vue.js, es momento de crear un bucket S3 para alojarlo. Mediante la consola de Amazon se crea un bucket S3 simple y se sube la carpeta resultada de ejecutar `'npm run build'`.

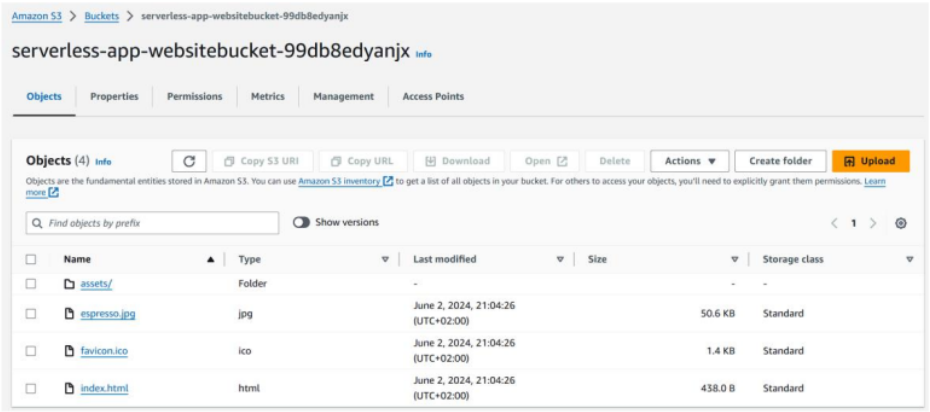

Figura 12. Bucket S3 con los archivos del frontend

Para la creación del bucket se usó las opciones por defecto, a excepción de la habilitación del versionado del bucket.

Para probar el frontend alojado en S3, aunque sin conexión al backend todavía, es necesario crear la distribución CloudFront. Para esto, se crea una distribución de CloudFront en donde se le indica el S3 bucket como origen. Además, se restringe el acceso del bucket S3 para que solo se pueda acceder a través de CloudFront. Esto genera una policy que debemos agregar al bucket. Además, se añade a la configuración que cuando soliciten la página con http, los redirija a https. También se configura la caché y la redirección de errores hacia index.html para que Vue.js lo maneje correctamente a través de su gestor de router.

Una vez que se tiene la distribución creada y configurada, se puede acceder a la aplicación web mediante la URL proporcionada por CloudFront.

Se puede observar en la Figura 13 que, efectivamente, se puede acceder a la página web. Sin embargo, debido a que la API aún no está creada, no es posible conectarse a ella, por lo cual se muestra el mensaje de error.

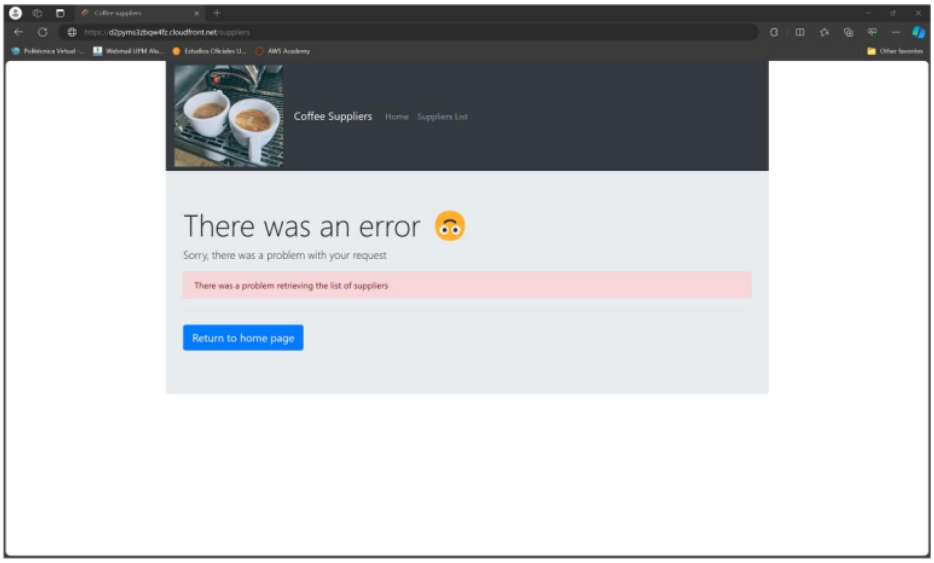

Figura 13. Página web accedida mediante la URL proporcionada por CloudFront

5.2.2 Base de datos

Para sustituir la base de datos usada en la aplicación original se usará DynamoDB, que es una base de datos serverless NoSQL. Aunque existe la posibilidad de crear una base de datos SQL serverless gracias a Aurora, se ha optado por escoger DynamoDB debido a que se apega más a la arquitectura serverless, debido a que es un servicio completamente administrado y altamente escalable. Además, DynamoDB cuenta con la capa gratuita de Amazon por siempre, en donde obtienes 25GB de almacenamiento y 25 unidades de capacidad de escritura y lectura, lo que es suficiente para administrar 200 millones de solicitudes al mes. Por el contrario, Aurora solo ofrece su capa gratuita durante los primeros 12 meses desde la creación de la cuenta AWS, con 750 horas al mes de uso de la base de datos y 20GB de almacenamiento.

Solo es necesario crear una tabla dentro de DynamoDB. DynamoDB guarda los datos en tablas, y dichas tablas guardan ítems, que a su vez almacenan atributos. Se creará una tabla con 2 unidades de capacidad de escritura y 2 de lectura y con el id del proveedor como 'Partition Key', o en este caso, clave primaria.

Con la tabla creada, es posible agregar ítems directamente desde la consola de Amazon, como se muestra en la Figura 14.

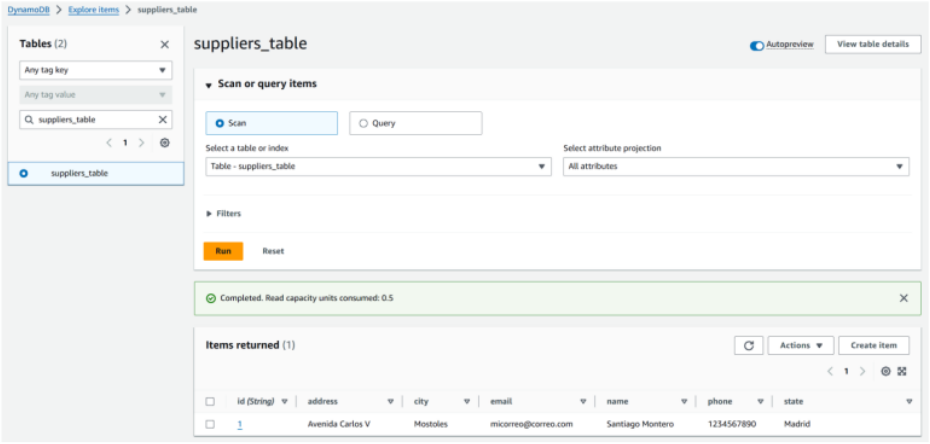

Figura 14. Tabla de DynamoDB con un item creado

5.2.3 Funciones Lambda

Para la creación de las funciones lambdas se tendrá en cuenta las solicitudes que se hacían en la aplicación original a cada una de las rutas. Cada una de estas solicitudes ahora apuntarán a una API, la cual a su vez apuntará a cada una de las funciones lambda para el procesamiento de la solicitud. Por lo tanto, se van a crear 5 funciones lambda que serán escritas en Python.

Cada función obtiene el nombre de la tabla de DynamoDB a través de una variable de entorno denominada TABLE_NAME. Haciendo uso de la AWS SDK para Python, llamada Boto3, se puede interactuar directamente con DynamoDB y hacer las operaciones correspondientes.

Las funciones se crearán en una carpeta con la estructura que se puede observar en la Figura 15, de donde luego se subirán a AWS Lambda para su despliegue y configuración de rol que permita la interacción con DynamoDB.

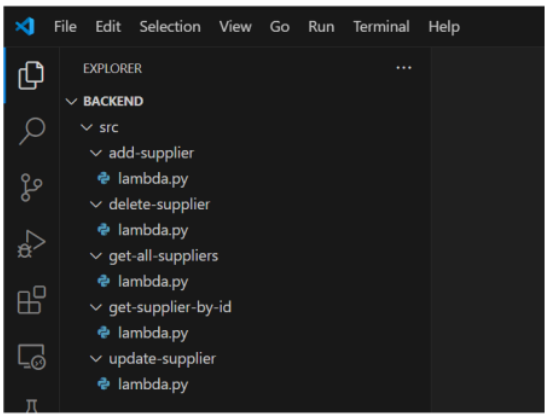

Figura 15. Estructura para la creación de las funciones lambda

Las funciones cumplen la funcionalidad descrita por su nombre.

- **Función add-supplier**

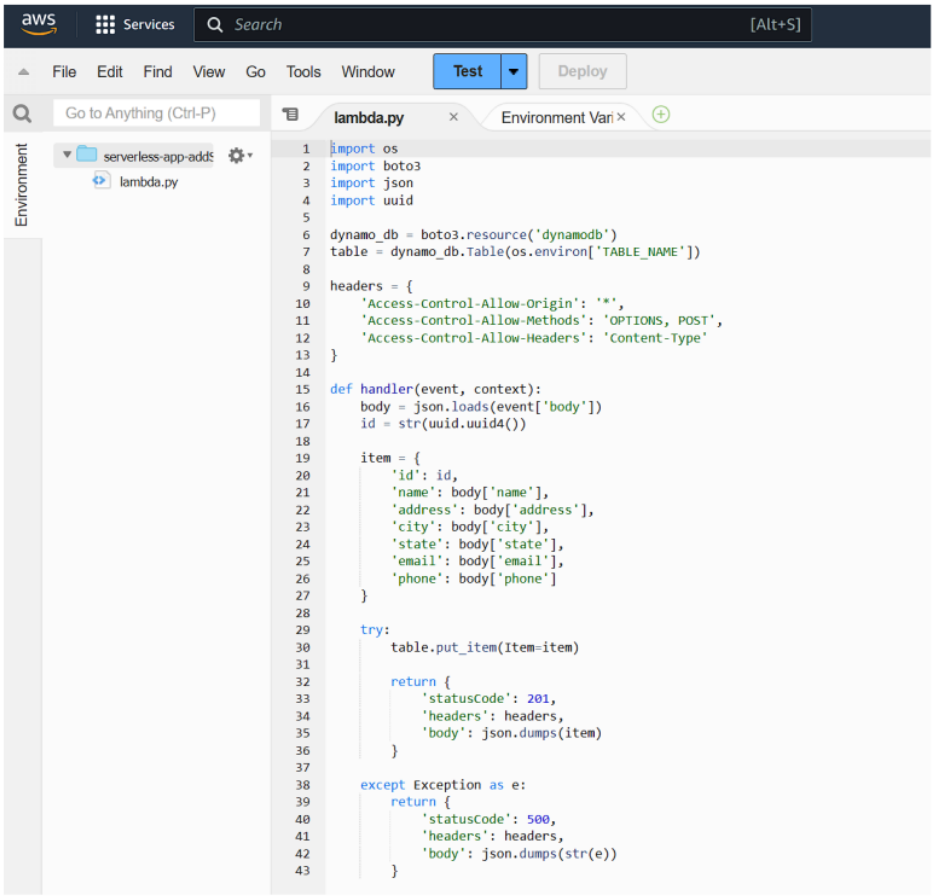

Figura 16. Implementación de función para añadir un proveedor

Esta función lambda se encarga de recibir la información de un nuevo proveedor y crear un nuevo item en la tabla de DynamoDB. Si todo va bien, devuelve el nuevo proveedor creado junto al id generado y un código de respuesta 201.

- **Función delete-supplier**

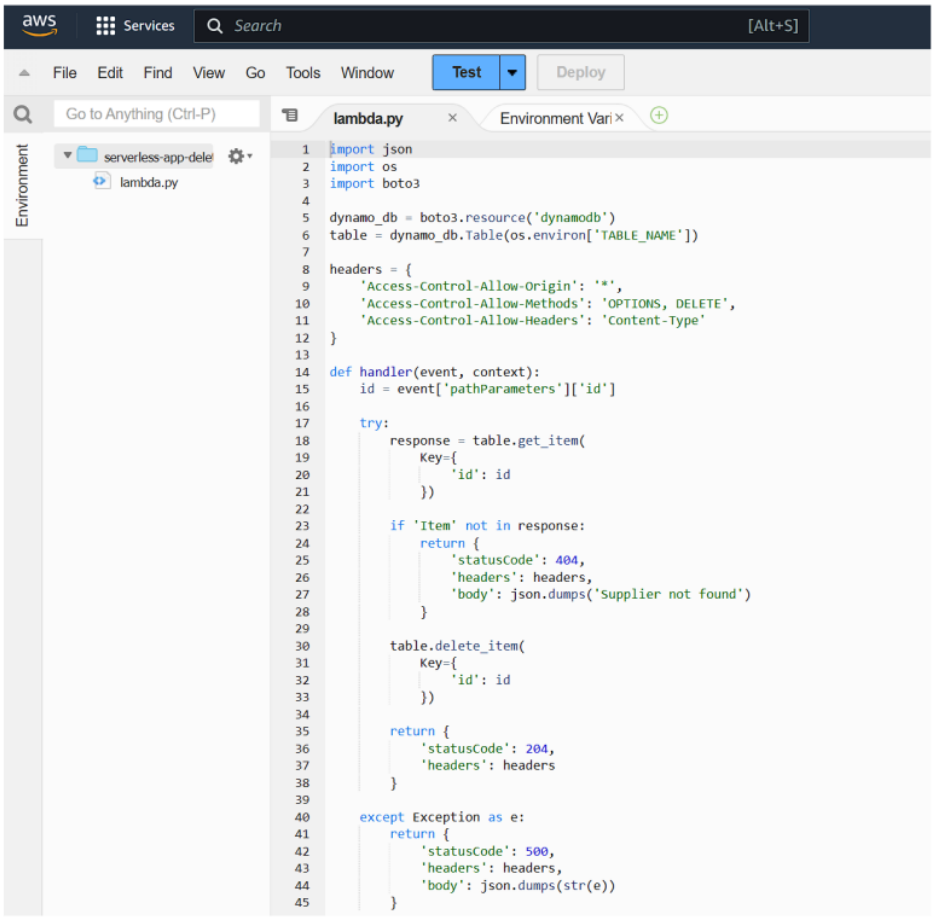

Figura 17. Implementación de función para eliminar un proveedor

Esta función lambda se encarga de eliminar un proveedor de la tabla de DynamoDB según el id pasado en el path de la solicitud. Si todo va bien, devuelve un código de respuesta 204.

- **Función get-supplier-by-id**

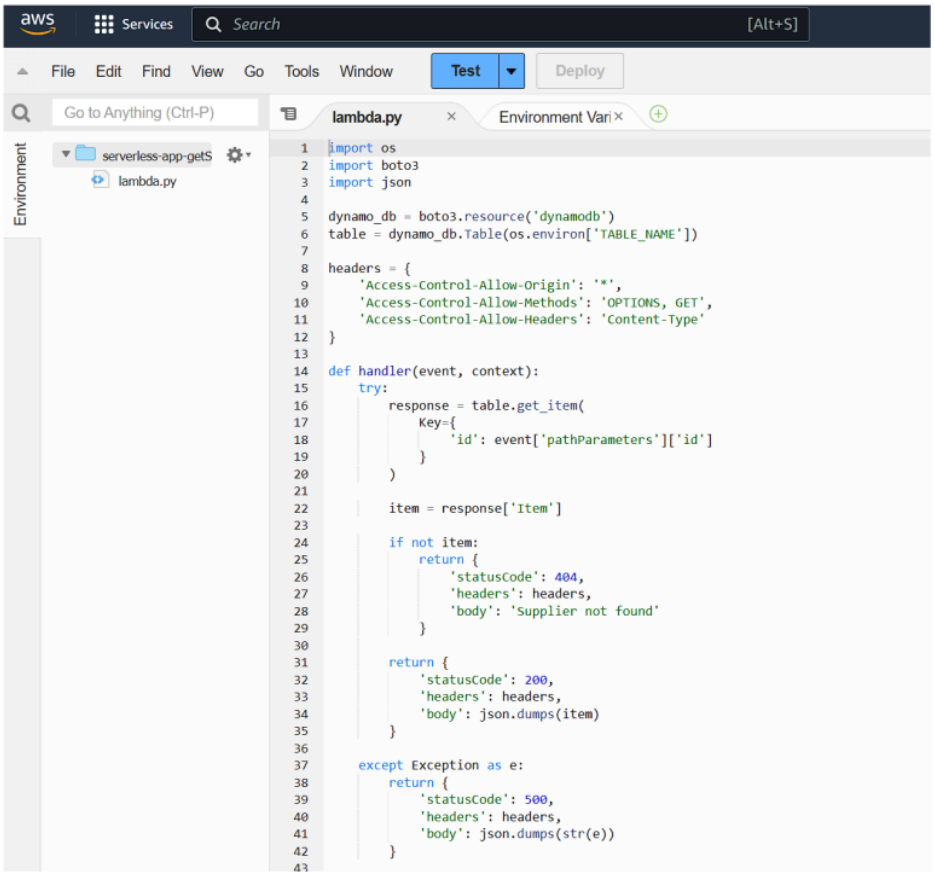

Figura 18. Implementación de función para obtener un proveedor por su id

Esta función lambda se encarga de obtener y devolver la información de un proveedor de la tabla de DynamoDB dado su id en el path de la solicitud. Si todo va bien, devuelve la información del proveedor solicitado junto con un código de respuesta 200.

- **Función update-supplier**

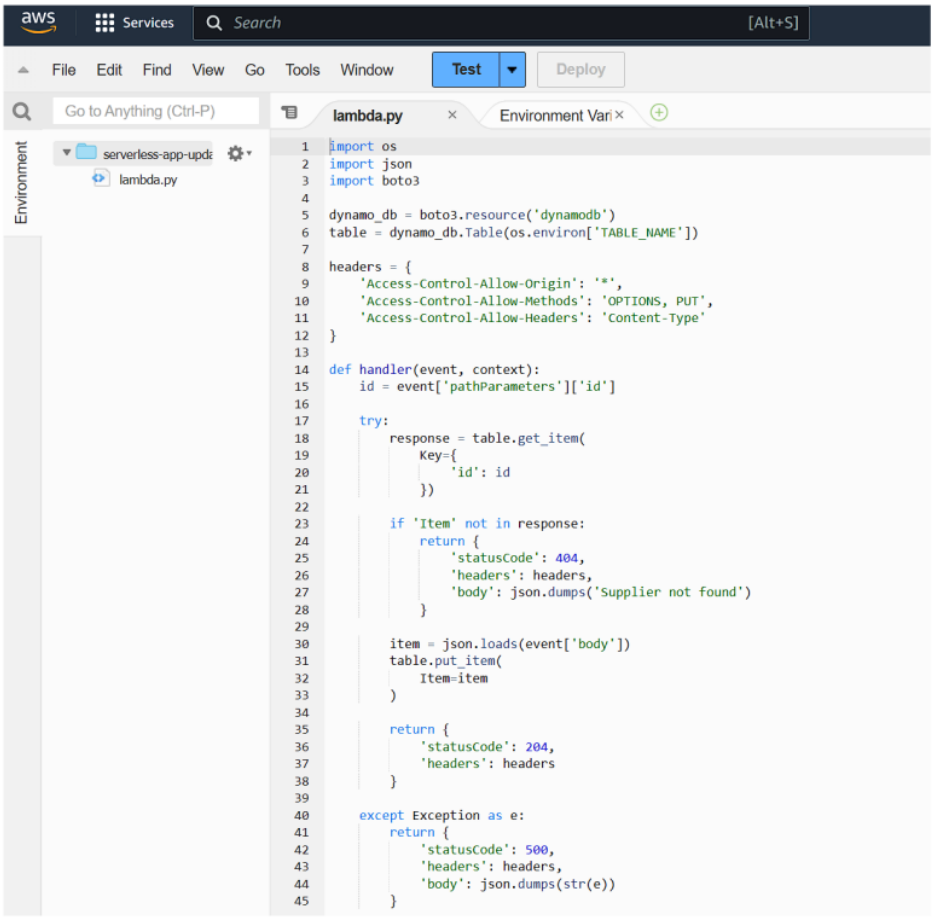

Figura 19. Implementación de función para actualizar un proveedor

Esta función lambda se encarga de actualizar la información de un proveedor en la tabla de DynamoDB con la nueva información pasada a través del cuerpo de la solicitud. Si todo va bien, devuelve un código de respuesta 204.

- **Función get-all-suppliers**

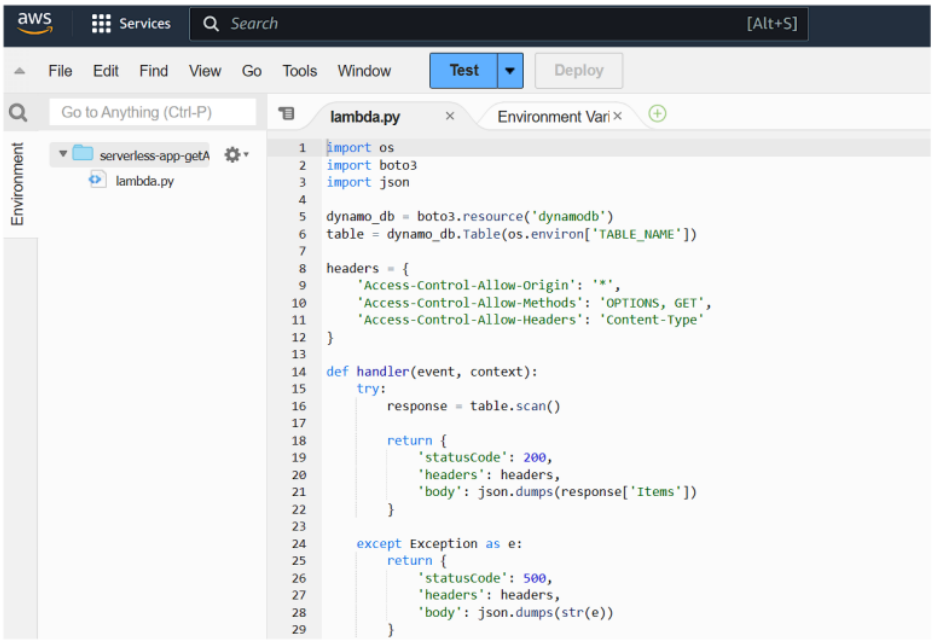

Figura 20. Implementación de función para obtener todos los proveedores

Esta función lambda se encarga de devolver una lista de todos los proveedores, junto con su información, existentes en la tabla de DynamoDB. Si todo va bien, devuelve todos los proveedores y un código de respuesta 200.

5.2.4 API

Para la creación de la REST API, se creará una API de tipo REST en API Gateway. Dentro de esta, se creará un recurso por cada uno de los que aparecen en la aplicación original con sus respectivos métodos, es decir, se crearán los siguientes recursos:

- /suppliers
  - GET
  - POST
  - OPTIONS (para el manejo de solicitudes preflight en CORS)

- /suppliers/{id}
  - DELETE
  - GET
  - PUT
  - OPTIONS (para el manejo de solicitudes preflight en CORS)

Cada recurso creado se conectará con su función lambda correspondiente para gestionar la solicitud. Una vez que cada función lambda está conectada a cada recurso, cada vez que se haga una solicitud a ese recurso, la función lambda será lanzada y se obtendrá una respuesta.

Para probar su funcionamiento se creará un despliegue de la API creada. Con la URL proporcionada por el despliegue de la API y con la ayuda de POSTMAN se comprobará el funcionamiento de esta.

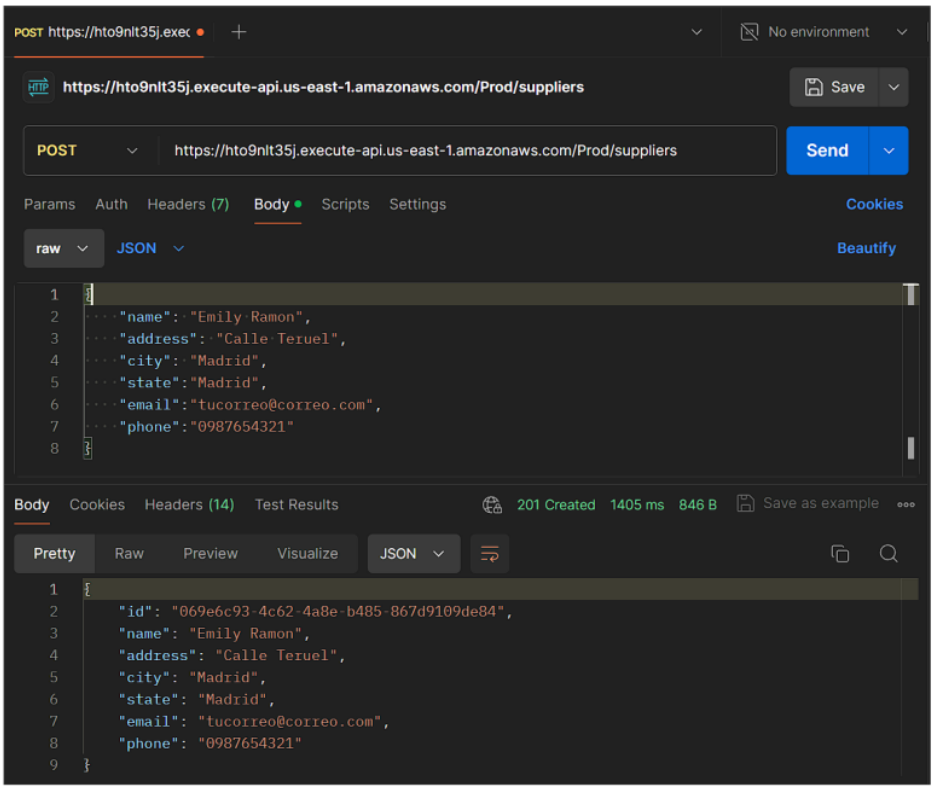

Figura 21. Solicitud POST hacia la API creada

Una vez que se comprueba el funcionamiento de la API, solo faltaría configurar una variable de entorno adecuada en el frontend para que pueda enviar solicitudes hacia la API.

Para actualizar el frontend, es necesario cargar nuevamente los archivos en el bucket S3, pero esta vez agregando un archivo .env, con una variable de entorno que contenga el endpoint de la API, antes de correr `'npm run build'`.

Con el frontend actualizado, finalmente se obtiene la aplicación monolítica entera desacoplada y desplegada en una arquitectura serverless. Accediendo nuevamente a la URL proporcionada por CloudFront se puede acceder a la aplicación web y comprobar que efectivamente el post realizado a través de POSTMAN se hizo efectivo. Es posible también agregar más proveedores, editar su información y eliminarlos.

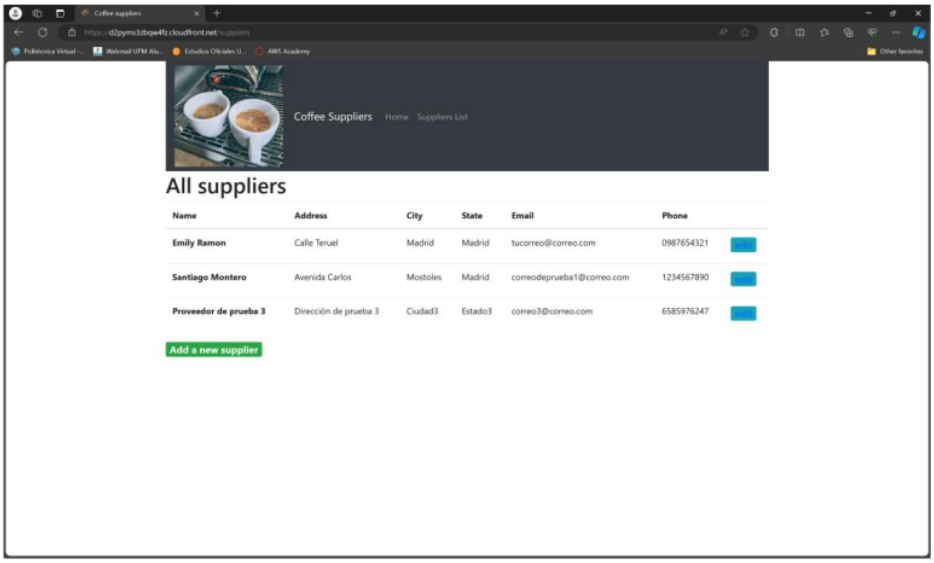

Figura 22. Aplicación web con 3 proveedores de prueba

5.3 Automatización de la Creación de la Infraestructura

Para la automatización de la creación de la infraestructura es necesario definir la infraestructura como código (IaC). Para ello, AWS ofrece herramientas para este fin. En este proyecto se ha optado por el uso de AWS SAM debido a que el desarrollo se trata de una aplicación serverless. AWS SAM permite definir aplicaciones serverless a través de plantillas con formato YAML. En estas plantillas la definición de, por ejemplo, APIs, funciones lambda o bases de datos es más simple debido al uso de una sintaxis simplificada comparándolo con una plantilla de 22oudFormation. Sin embargo, al final AWS CloudFormation se encarga de la realización de las tareas necesarias para el despliegue de los servicios definidos en la plantilla de AWS SAM.

Por lo tanto, para lograr la automatización de la creación de la infraestructura, es necesario definir una plantilla SAM con todos los servicios que se han creado en los apartados anteriores y su configuración correspondiente.

El nombre de la plantilla SAM, en este caso, será `'template.yaml'`.

A continuación, se muestra la definición en la plantilla de los servicios creados anteriormente. Se mostrarán colapsados los campos más largos y con menos relevancia.

- Rol que permite la interacción con DynamoDB y logs de CloudWatch a las funciones lambda.

```yaml
LambdaExecutionRole:
  Type: AWS::IAM::Role
  Properties:
    AssumeRolePolicyDocument: ···
    Policies:
      - PolicyName: LambdaExecutionPolicy
        PolicyDocument:
          Version: '2012-10-17'
          Statement:
            - Effect: Allow
              Action:
                - logs:CreateLogGroup
                - logs:CreateLogStream
                - logs:PutLogEvents
                - dynamodb:PutItem
                - dynamodb:GetItem
                - dynamodb:DeleteItem
                - dynamodb:Scan
                - dynamodb:UpdateItem
              Resource: '*'
```

Figura 23. Definición de LambdaExecutionRole

- API de API Gateway con los métodos permitidos y un despliegue con nombre Prod. Más adelante se definen los recursos de esta API mediante la definición de las funciones lambda.

```yaml
CoffeeSuppliersApi:
  Type: AWS::Serverless::Api
  Properties:
    StageName: Prod
    Cors:
      AllowMethods: '''OPTIONS, POST, GET, PUT, DELETE'''
      AllowHeaders: '''Content-Type'''
      AllowOrigin: '''*'''
```

Figura 24. Definición de CoffeeSuppliersApi

- Todas las funciones lambda se definen de la misma manera que la indicada en la Figura 26, en donde asumen el rol creado antes, especifican el origen de su código, se crean sus variables de entorno para el nombre de la tabla de DynamoDB y se define el recurso de la API con el que la función será lanzada a ejecución.

```yaml
addSuplierFunction:
  Type: AWS::Serverless::Function
  Properties:
    CodeUri: backend/src/add-supplier/
    Handler: lambda.handler
    Runtime: python3.11
    Role: !GetAtt LambdaExecutionRole.Arn
    Environment:
      Variables:
        TABLE_NAME: !Ref SuppliersTable
    Events:
      Api:
        Type: Api
        Properties:
          Path: /suppliers
          Method: POST
          RestApiId: !Ref CoffeeSuppliersApi
```

Figura 25. Definición de función lambda

- Tabla de DynamoDB para el almacenamiento de la información de los proveedores. Se define id como su clave primaria y se configura las unidades de capacidad de escritura y lectura a 2.

```yaml
SuppliersTable:
  Type: AWS::Serverless::SimpleTable
  Properties:
    PrimaryKey:
      Name: id
      Type: String
    ProvisionedThroughput:
      ReadCapacityUnits: 2
      WriteCapacityUnits: 2
```

Figura 26. Definición de SuppliersTable

- Bucket S3 para el alojamiento de la aplicación web desarrollada en Vue.js junto con su Policy para solo permitir el acceso al bucket mediante CloudFront.

```yaml
WebsiteBucket:
  Type: AWS::S3::Bucket
  DeletionPolicy: Retain
  Properties:
    VersioningConfiguration:
      Status: Enabled

WebsiteBucketPolicy:
  Type: AWS::S3::BucketPolicy
  Properties:
    Bucket: !Ref WebsiteBucket
    PolicyDocument:
      Version: '2012-10-17'
      Id: PolicyForCloudFrontPrivateContent
      Statement:
        - Sid: AllowCloudFrontServicePrincipal
          Effect: Allow
          Principal:
            Service: cloudfront.amazonaws.com
          Action: s3:GetObject
          Resource: !Sub arn:aws:s3:::${WebsiteBucket}/*
          Condition:
            StringEquals:
              AWS:SourceArn: !Sub arn:aws:cloudfront::${AWS::AccountId}:distribution/${CloudFrontDistribution}
```

Figura 27. Definición de WebsiteBucket y su Policy

- Control de acceso de origen para la distribución de CloudFront que permite que solo la distribución de CloudFront tenga acceso al origen, que es el bucket S3.

```yaml
CloudFrontOriginAccessControl:
  Type: AWS::CloudFront::OriginAccessControl
  Properties:
    OriginAccessControlConfig:
      Name: !Sub ${WebsiteBucket} OAC
      OriginAccessControlOriginType: s3
      SigningBehavior: always
      SigningProtocol: sigv4
```

Figura 28. Definición de CloudFrontOriginAccessControl

- Distribución de CloudFront con WebsiteBucket como origen. Se define el objeto por defecto, index.html. Se configura la redirección de errores para que Vue.js pueda gestionar adecuadamente las rutas. Se configura las opciones de caché y la opción para redirigir las solicitudes desde HTTP a HTTPS. Por último, se configura PriceClass que indica que solo se usaran las ubicaciones de borde de Europa, Israel, Turquía, Estados Unidos, México y Canadá.

```yaml
CloudFrontDistribution:
  Type: AWS::CloudFront::Distribution
  Properties:
    DistributionConfig:
      Origins:
        - DomainName: !GetAtt WebsiteBucket.RegionalDomainName
          Id: myS3Origin
          OriginAccessControlId: !GetAtt CloudFrontOriginAccessControl.Id
          S3OriginConfig:
            OriginAccessIdentity: ''
      Enabled: true
      DefaultRootObject: index.html
      CustomErrorResponses:
        - ErrorCode: 403
          ErrorCachingMinTTL: 3600
          ResponseCode: 200
          ResponsePagePath: /index.html
      HttpVersion: http2
      DefaultCacheBehavior:
        AllowedMethods:
          - DELETE
          - GET
          - HEAD
          - OPTIONS
          - PATCH
          - POST
          - PUT
        CachedMethods:
          - GET
          - HEAD
        TargetOriginId: myS3Origin
        ForwardedValues:
          QueryString: false
          Cookies:
            Forward: none
        ViewerProtocolPolicy: redirect-to-https
        MinTTL: 0
        DefaultTTL: 3600
        MaxTTL: 86400
      PriceClass: PriceClass_100
      ViewerCertificate:
          CloudFrontDefaultCertificate: true
```

Figura 29. Definición de CloudFrontDistribution

Con el template creado es posible realizar el despliegue de este, mediante AWS SAM. Para efectuarlo se ejecuta primero `'sam build'`. A continuación, se ejecuta el comando `'sam deploy –guided'`, el cual realiza una serie de preguntas para establecer el archivo de configuración de este despliegue. En este caso, las respuestas a las preguntas fueron las siguientes:

```
Stack Name [sam-app]: serverless-app
AWS Region [us-east-1]:
#Shows you resources changes to be deployed and require a 'Y' to initiate deploy
Confirm changes before deploy [Y/n]:
#SAM needs permission to be able to create roles to connect to the resources in
your template
Allow SAM CLI IAM role creation [Y/n]: n
Capabilities [['CAPABILITY_IAM']]:
#Preserves the state of previously provisioned resources when an operation fails
Disable rollback [y/N]:
addSuplierFunction has no authentication. Is this okay? [y/N]: y
deleteSupplierFunction has no authentication. Is this okay? [y/N]: y
getAllSuppliersFunction has no authentication. Is this okay? [y/N]: y
getSupplierByIdFunction has no authentication. Is this okay? [y/N]: y
updateSupplierFunction has no authentication. Is this okay? [y/N]: y
Save arguments to configuration file [Y/n]:
SAM configuration file [samconfig.toml]:
SAM configuration environment [default]:
```

Esto genera un changeset, el cual es necesario confirmar para que inicie la creación, en un stack de CloudFormation, de los recursos definidos en el template.

Cuando CloudFormation termina de crear el stack, la infraestructura para la aplicación está desplegada. Sin embargo, para poder acceder a la aplicación web hay que subir manualmente los archivos de la web al bucket creado por el stack de CloudFormation.

5.4 Definición del Pipeline CI/CD

Para poder automatizar el despliegue del template anterior y el despliegue del frontend al bucket S3 cada vez que se realicen cambios en el repositorio de la aplicación es necesario la creación de un pipeline CI/CD, que será creado con el servicio de AWS CodePipeline junto con AWS CodeBuild.

Para la realización de esta tarea primero es necesario tener el código de la aplicación en un repositorio. En este caso, se usará un repositorio creado en AWS CodeCommit llamado 'serverless'. Segundo, es necesaria la creación de otro template que defina los recursos necesarios para el pipeline.

AWS SAM ofrece una plantilla inicial para la creación de un pipeline a través del comando 'sam pipeline init –bootstrap'. Esta plantilla inicial contiene la definición de recursos que son necesarios para la ejecución del pipeline, por ejemplo, los roles adecuados y un bucket S3 para guardar los artefactos que se generen en cada fase del pipeline y sean necesarios a la entrada de la siguiente fase. Se hará uso de dicha plantilla, aunque será modificada casi por completo para ajustarlo al caso de uso de este proyecto.

El pipeline tendrá 3 fases principales, como se indicó en el diagrama presentado en el Capítulo 4. La primera fase se denomina 'Source', en la que se obtiene el código del repositorio. La segunda fase se denomina 'BuildAndPackage', en la que mediante un proyecto de CodeBuild se ejecuta 'sam build' para construir y compilar las funciones lambda, y demás artifactos necesarios, y 'sam package' para empaquetar los artifactos y guardarlos en un bucket S3 para ser usados por la siguiente fase del pipeline. La tercera fase es la última de este pipeline, denominada 'Deploy'. En esta fase se ejecuta 'sam deploy' con los argumentos necesarios para desplegar el template de la arquitectura serverless. También se ejecuta un script para desplegar el frontend al bucket S3 que aloja el frontend.

Este pipeline contará con una fase extra para actualizar el propio recurso de pipeline. Esto quiere decir que, si se modifica el pipeline y se suben los cambios al repositorio, el pipeline iniciará y desplegará los cambios que se hayan hecho al mismo pipeline. Para entenderlo mejor, si se cambian las fases del pipeline y se agrega por ejemplo una fase de prueba, el pipeline después de la fase 'Source' se modificará y agregará la nueva fase de prueba para ser usada inmediatamente después.

En la plantilla en la que se definirá el pipeline, también se definirá la creación de un bucket para el almacenamiento de los artefactos necesarios para desplegar la infraestructura serverless, además de dos roles para la ejecución adecuada del pipeline.

Una vez que se tiene la plantilla del pipeline definida, hay que subir todo al repositorio y desplegar la plantilla del pipeline con 'sam deploy -t codepipeline.yaml --capabilities=CAPABILITY_IAM --stack-name nombre_stack_pipeline'

Cuando el stack se termine de desplegar, el pipeline se activará automáticamente y creara la infraestructura para la aplicación y el despliegue del frontend al bucket S3. Esto quiere decir que cuando el pipeline termine de ejecutarse, la aplicación quedará lista para ser accedida mediante la URL proporcionada por CloudFront.

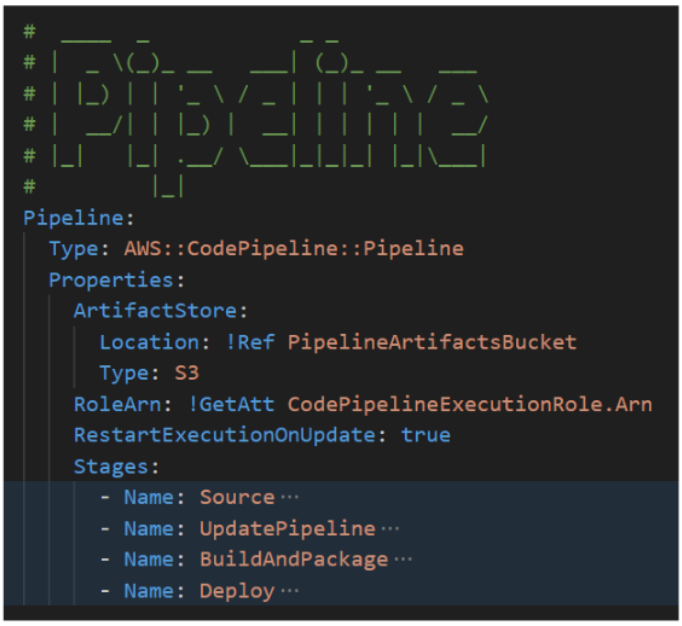

Figura 30. Definición del pipeline

6 Resultados y conclusiones

Con el desacoplamiento de la aplicación monolítica y su transformación en una aplicación con arquitectura serverless se han atacado efectivamente los puntos de problema principales mencionados en el análisis de la aplicación monolítica.

A continuación, se detallan y evalúan las ventajas que otorga el nuevo enfoque frente al antiguo.

- **Disponibilidad:** Debido a que los servicios usados en la construcción de la arquitectura serverless son completamente administrados por Amazon, son altamente disponibles. En contraste, una instancia EC2 por si sola no es altamente disponible debido a varias razones, entre las que se puede nombrar la ubicación en una sola zona de disponibilidad dentro de la región en la que se encuentre, posibles fallos de memoria, red o hardware que no puede manejar automáticamente y la posible necesidad de realizar mantenimientos de hardware o software por los cuales se interrumpe el servicio.

- **Escalabilidad:** Si existen picos de trabajo muy altos, la nueva arquitectura no tiene ningún problema en escalar automáticamente y manejar la alta demanda, gracias nuevamente al uso de servicios serverless. Por otro lado, la arquitectura antigua no sería capaz ni si quiera de escalar por si sola, lo que podría ocasionar cuellos de botella y perjudicar sustancialmente al rendimiento.

- **Rendimiento:** Un aspecto de mejora también es el rendimiento. Es evidente que gracias a la escalabilidad que tiene la nueva arquitectura, el rendimiento mejora notablemente. Pero, además, gracias a CloudFront la distribución de la página web se hace desde los servidores más cercanos al cliente. Esto junto con la caché que proporciona CloudFront hace que incluso en los momentos con más alta demanda, la nueva aplicación pueda tener un buen rendimiento y tiempos de respuesta muy bajos.

- **Seguridad:** Todos los datos almacenados en DynamoDB y buckets S3 son encriptados automáticamente en reposo. Además, CloudFront tiene una protección integrada contra ataques de denegación de servicio (DDoS) que la aplicación monolítica original no tiene.

- **Productividad y nuevas funcionalidades:** Gracias a la creación del pipeline, solo es necesario hacer los cambios que consideres oportunos en el código y subirlo a tu repositorio. Inmediatamente, el pipeline se activa y empieza el proceso de despliegue. Es importante mencionar que en el proceso de despliegue solo se actualiza o crea lo cambios hechos en el repositorio, no la arquitectura entera. Esto permite enfocarte en tu lógica de negocio y no desperdiciar tiempo en el mantenimiento o creación de infraestructura. Además, gracias también a la arquitectura, se pueden implementar o probar nuevas funcionalidades fácil y

rápidamente gracias a poder replicar el stack con el que estás trabajando. Abre un mundo de posibilidades.

Para hacer una comparación de costos entre las dos arquitecturas, hay que plantear dos situaciones y asumir la suposición hecha en la estimación de costos.

La primera situación, es asumir que la cuenta de Amazon en la que se desplegará la arquitectura es nueva, lo cual permite beneficiarte de ciertos servicios de la capa gratuita de Amazon que solo tienen validez los 12 primeros meses.

Asumiendo esto, el costo aproximado de la arquitectura serverless junto con el pipeline, es de 0,84 USD durante los 12 primeros meses, como fue mencionado en la estimación de costo.

Por el otro lado, manteniendo la arquitectura antigua con todas las desventajas que implica, los 12 primeros meses tendrían un valor de 0 USD debido a la capa gratuita que se ofrece para Amazon RDS y EC2.

La segunda situación, es asumir que la cuenta de Amazon en la que se desplegará la arquitectura NO es nueva.

En este caso, los únicos servicios de la arquitectura serverless que tienen un límite de 12 meses en sus capas gratuitas, son Amazon S3 y API Gateway. Por lo cual, habría que sumar 4,56 USD en concepto de API Gateway, y otros 0,08 USD en concepto de API Gateway. Como resultado final, los 12 primeros meses cuestan 5,47 USD.

En cambio, para la arquitectura antigua hay que sumar 132,72 USD en concepto de Amazon RDS, y 72,72 en concepto de EC2. Como resultado final, los 12 primeros meses cuestan 205,44 USD.

Es una diferencia abismal. Como conclusión, pienso que es mucho mejor migrar hacia la arquitectura serverless aunque el primer año con cuentas nuevas cueste 0,84 USD más caro que la arquitectura monolítica.

La nueva aplicación tiene varias posibilidades de mejora, como usar Amazon Cognito para crear un login en la página web o control de acceso a la API mediante Cognito. Otra posible mejora es agregar a la arquitectura AWS WAF, lo cual mejoraría aún más la seguridad gracias a la posibilidad de creación de reglas que controlen el tráfico de bots o los ataques más comunes.

7 Análisis de Impacto

7.1 Impacto Personal

La realización de este proyecto me ha ayudado a profundizar mis conocimientos sobre AWS en general, pero más específicamente en aquellos servicios que forman parte de las arquitecturas serverless. Entre estos servicios están AWS Lambda, Amazon CloudFront, DynamoDB y API Gateway. Además, he tenido la oportunidad de aprender cómo se automatiza la creación de infraestructuras y lo beneficioso que puede llegar a ser. Aprendí también más sobre CI/CD, y descubrí los beneficios prácticos que te ofrece el despliegue automatizado. Por último, aprendí a diseñar y realizar una estimación de coste de una arquitectura.

7.2 Impacto Empresarial

Como se mencionó anteriormente en las conclusiones, a nivel de empresa el impacto es muy grande. Esto es porque obtienes escalabilidad de tu producto, lo cual mejora el rendimiento, que, a su vez, mejora la experiencia de usuario. También multiplica tu productividad al no tener que encargarte de mantenimiento ni de la creación manual de infraestructura. Lo cual permite que la empresa se centré en su lógica de negocio.

7.3 Impacto Social y Económico

La migración hacia arquitecturas serverless puede permitir la creación de oportunidades laborales en otras empresas, que a su vez pueden ofrecer formación en el campo de computación en la nube.

7.4 Impacto Medioambiental

Los servicios serverless están diseñados y optimizados para ser altamente eficientes en el consumo de recursos. Debido a su escalabilidad en momentos de mucho tráfico, evitan el desperdicio de recursos con lo cual se disminuye la huella de carbono.

7.5 Relación con los Objetivos de Desarrollo Sostenible

El proyecto se relaciona con varios objetivos de Desarrollo Sostenible.

- ODS 7 – Energía Asequible y No Contaminante: Amazon se ha comprometido a operar sus centros con energía 100% renovable.
- ODS 12 – Producción y Consumo Responsables: Los servicios serverless optimizan los recursos computacionales y reducen la necesidad de infraestructura física, lo cual ayuda a una producción más sostenible y responsable.

8 Bibliografía

[1] Red Hat, "What is CI/CD?," Red Hat, 2023. [Online]. Available: https://www.redhat.com/es/topics/devops/what-is-ci-cd. [Accessed: May, 2024].

[2] Amazon Web Services, "CI/CD for 5G Networks on AWS," AWS, 2023. [Online]. Available: https://docs.aws.amazon.com/whitepapers/latest/cicd_for_5g_networks_on_aws/cicd-on-aws.html. [Accessed: May, 2024].

[3] Amazon Web Services, "AWS CodeCommit," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codecommit/. [Accessed: May, 2024].

[4] Amazon Web Services, "AWS CodeBuild," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codebuild/. [Accessed: May, 2024].

[5] Amazon Web Services, "AWS CodePipeline," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codepipeline/. [Accessed: May, 2024].

[6] Amazon Web Services, "AWS CodeDeploy," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/codedeploy/. [Accessed: May, 2024].

[7] Amazon Web Services, "AWS CloudFormation," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/cloudformation/. [Accessed: May, 2024].

[8] Amazon Web Services, "Arquitecturas serverless," AWS, 2024. [Online]. Available: https://aws.amazon.com/es/lambda/serverless-architectures-learn-more/. [Accessed: May, 2024].

[9] Amazon Web Services, "Decomposing monoliths into microservices," AWS Documentation. [Online]. Available: https://docs.aws.amazon.com/prescriptive-guidance/latest/modernization-decomposing-monoliths/welcome.html. [Accessed: May, 2024].

[10] Wikipedia, "Domain-driven design," Wikipedia, 2024. [Online]. Available: https://en.wikipedia.org/wiki/Domain-driven_design. [Accessed: May, 2024].

[11] Red Hat, "What is Blue-Green Deployment?," Red Hat, 2024. [Online]. Available: https://www.redhat.com/es/topics/devops/what-is-blue-green-deployment. [Accessed: May 30, 2024].

TFG.pdf

ORIGINALITY REPORT

10%	9%	1%	4%
SIMILARITY INDEX	INTERNET SOURCES	PUBLICATIONS	STUDENT PAPERS

PRIMARY SOURCES

1	Submitted to Universidad Politécnica de Madrid Student Paper	2%
2	Submitted to Birmingham Metropolitan College Student Paper	1%
3	oa.upm.es Internet Source	1%
4	www.celestialchronicle.com Internet Source	1%
5	docs.aws.amazon.com Internet Source	1%
6	aws.amazon.com Internet Source	1%
7	qiita.com Internet Source	1%
8	1library.co Internet Source	<1%
9	repositorio.uam.es	

	Internet Source	<1%
10	Submitted to Universitat Politècnica de València Student Paper	<1%
11	Submitted to Fakultet elektrotehnike i računarstva / Faculty of Electrical Engineering and Computing Student Paper	<1%
12	Submitted to Colorado State University, Global Campus Student Paper	<1%
13	rua.ua.es Internet Source	<1%
14	Submitted to Saint Louis University Student Paper	<1%
15	es.slideshare.net Internet Source	<1%
16	Submitted to Narrabundah College Student Paper	<1%
17	www.fundacionauna.org Internet Source	<1%
18	issuu.com Internet Source	<1%
19	www.elastic.co	

	Internet Source	<1%
20	fourweekmba.com Internet Source	<1%
21	repositorio.urp.edu.pe Internet Source	<1%
22	www.agrocolor.es Internet Source	<1%
23	www.dte.uvigo.es Internet Source	<1%
24	www.ni.com Internet Source	<1%
25	argentina.linefeed.org Internet Source	<1%
26	catalonica.bnc.cat Internet Source	<1%
27	linuxalbacete.org Internet Source	<1%
28	scc.coomeva.com Internet Source	<1%
29	www-142.ibm.com Internet Source	<1%
30	www.coursehero.com Internet Source	<1%

31	www.inf.udec.cl Internet Source	<1 %
32	www.pergaminovirtual.com.ar Internet Source	<1 %
33	bibliotecadigital.exactas.uba.ar Internet Source	<1 %
34	library.e.abb.com Internet Source	<1 %
35	www.carm.es Internet Source	<1 %
36	www.citrixlac.com Internet Source	<1 %
37	www.dit.upm.es Internet Source	<1 %
38	www.dropbox.com Internet Source	<1 %
39	www.elsiglo.com Internet Source	<1 %
40	www.google.nl Internet Source	<1 %
41	www.portalamericas.com.ni Internet Source	<1 %
42	www.slideshare.net Internet Source	<1 %

43 "El estado mundial de la agricultura y la alimentación 2019", Food and Agriculture Organization of the United Nations (FAO), 2019

Publication

<1%

Exclude quotes	Off	Exclude matches	Off
Exclude bibliography	Off		

Este documento esta firmado por

	**Firmante**	CN=tfgm.fi.upm.es, OU=CCFI, O=ETS Ingenieros Informaticos - UPM, C=ES
	Fecha/Hora	Mon Jun 03 22:37:25 CEST 2024
	Emisor del Certificado	EMAILADDRESS=camanager@etsiinf.upm.es, CN=CA ETS Ingenieros Informaticos, O=ETS Ingenieros Informaticos - UPM, C=ES
	Numero de Serie	561
	Metodo	urn:adobe.com:Adobe.PPKLite:adbe.pkcs7.sha1 (Adobe Signature)