Diseño y creación de la infraestructura cloud y el software para el control de ocupación de puntos de carga para coches eléctricos

Jesús Jover, Carlos Medrano, Joaquín Tárraga

Resumen— **Este artículo explica cómo se ha diseñado, desplegado y construido un sistema de control de ocupación de puntos de carga para coches eléctricos en electrolineras. El objetivo es que el usuario pueda conocer si un punto de carga está siendo ocupado físicamente, esté o no cargando.**

Palabras clave— **Cloud, AWS, Lambda, DynamoDB, EC2, Terraform, HTTP, Raspberry Pi, Cognito, Load Balancer**

I. Introducción

Este trabajo pretende dar solución a una de los problemas más comunes a día de hoy de los propietarios de coches eléctricos: conocer si un punto de carga está ocupado.

A día de hoy ya existen sistemas que hacen esto: los propios puntos de carga suelen estar conectados continuamente con Internet, lo que les permite informar si existe un vehículo conectado al cargador. Sin embargo, un coche puede estar emplazado en un punto de carga y no estar conectado a él. De esta forma, si un propietario de coche eléctrico desea cargar su vehículo, consultando el estado del punto de carga verá que está disponible; no obstante, al llegar a él se encontrará con que físicamente hay un coche ocupando el lugar, lo que le lleva a la imposibilidad de realizar la carga.

Como ya sabemos, los coches eléctricos tienen una capacidad de carga limitada, que unido al hecho de la poca cantidad de puntos de carga disponibles hoy en día, puede hacer que el propietario recorra decenas de kilómetros, y que, a su llegada, se encuentre con ningún puesto disponible. Esto se agrava cuando, una vez en el punto de carga sin ningún puesto disponible, no tiene autonomía para desplazarse a otro punto de carga, viéndose en la necesidad de esperar o de buscar fuentes de alimentación alternativas.

Para ofrecer una solución, este trabajo explica la creación de la infraestructura cloud necesaria: la base de datos, los micro-servicios, un servidor virtual y la red subyacente. También ofrece detalle sobre el software ejecutado en esta arquitectura: la página web de control, y los end-points HTTP donde los puntos de carga y los coches eléctricos consulta información. Todos estos temas serán tratados en la Sección II. Finalmente, en la Sección III se presentan algunas pruebas de funcionamiento.

El código con el prototipo del sistema se encuentra disponible en el siguiente repositorio de GitHub: `https://github.com/carlosmn25/ABP-PyGITIC`. No obstante, no todos los módulos descritos en este informe se encuentran implementados en el mismo, dado que aquí se describe la arquitectura del sistema completo, pero no se ha realizado la implementación completa. Los módulos que se mencionan en este informe que todavía no han sido implementados son:

1. Backup periódico en S3.
2. Autenticación con Cognito.
3. Balanceo de carga en las instancias EC2.
4. Funcionamiento de cámara en la Raspberry Pi para lectura de matrículas.
5. Procedimiento de reservas de puntos de carga mediante matrícula.

Todos estos módulos, así como los que han sido implementados, serán explicados en la Sección II.

II. Componentes del sistema

A. Arquitectura general

La arquitectura propuesta para implementar el sistema se puede observar en la Figura 1. A nivel general, podemos separarla en 4 partes principales:

1. **Almacenamiento en DynamoDB**. Se encargará de registrar toda la información necesaria para el funcionamiento del sistema.
2. **Backups en S3**. Se realizarán copias de seguridad automáticas periódicamente para garantizar la integridad y disponibilidad de los datos.
3. **Actualización del estado de un punto de carga**. Mediante un dispositivo IoT en edge computing, el estado de los puntos de carga se actualizará automáticamente.
4. **Consulta de datos**. Devolverá información sobre las electrolineras y puntos de carga para usuarios del sistema.

A continuación, se describirá el funcionamiento de cada una de estas partes del sistema. Más concretamente, en la Sección II-A.1 se describe el almacenamiento en DynamoDB. La Sección II-A.2 desarrolla la realización de copias de seguridad periódicas en S3. La arquitectura del módulo de actualización de estado de los puntos de carga se detalla en la Sección II-A.3. Finalmente, la Sección II-A.4 desarrolla el funcionamiento del módulo de consulta de datos.

A.1 DynamoDB

Para el almacenamiento de los datos de forma persistente se ha seleccionado la base de datos no relacional DynamoDB, disponible en Amazon Web Services.

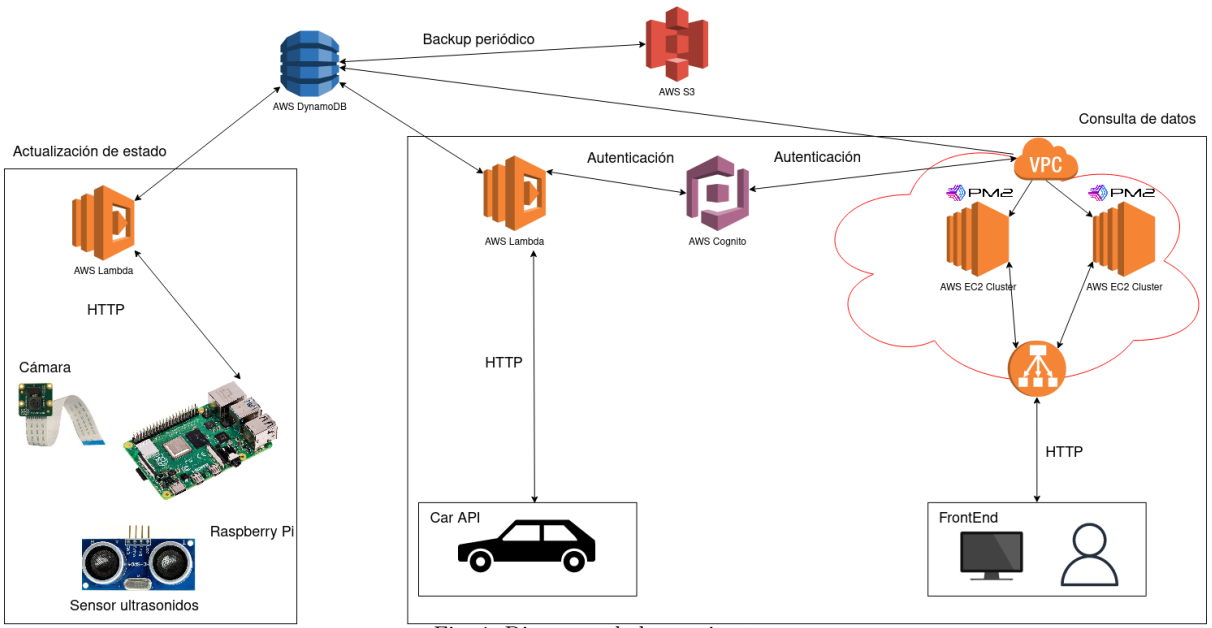

Fig. 1: Diagrama de la arquitectura

DynamoDB es una base de datos altamente escalable y completamente administrada, ofreciendo una solución perfecta para nuestro caso de uso. Su arquitectura sin servidores y su capacidad de escalar automáticamente según las demandas de tráfico hacen que sea una opción ideal para aplicaciones de alta disponibilidad y rendimiento. Al utilizar una infraestructura distribuida, DynamoDB garantiza la replicación y la tolerancia a fallos, lo que proporciona una mayor durabilidad y confiabilidad de los datos. Además, su modelo de datos flexible y sin esquemas permite adaptarse fácilmente a cambios en los requisitos de la aplicación.

Para almacenar los datos en DynamoDB se ha optado por la creación de cuatro tablas diferentes:

- La tabla **Electrolineras** que mantiene la información de las electrolineras como su nombre, ubicación, su estado (abierto, cerrado, etc), así como el número de puntos de carga disponibles.
- La tabla **PuntosCarga**, que mantiene la información de los puntos de carga, como la electrolinera donde se encuentra, el número dentro de la misma, su estado actual y su potencia de carga.
- La tabla **Estados** que almacena cada cambio de estado en cada uno de los puntos de carga. Se almacena información como el estado (sin carga/en carga/fuera de servicio), el momento en el que se ha producido el cambio, y la matrícula del coche en caso de estar en carga. Esta tabla será explicada en más detalle en la Sección II-D.
- La tabla **Estadísticas** que almacena información del estado de la red en un momento indicado, como el número de puntos sin carga, en carga, fuera de servicio y totales.

La configuración establecida para las tablas DynamoDB es la siguiente:

- **billing_mode** = "PROVISIONED". De esta forma, evitaremos que se cobre por cada petición efectuada, como ocurriría si establecemos la opción a "PAY_PER_REQUEST".
- **read_capacity** = 1. En DynamoDB, las unidades de lectura aprovisionadas representan la cantidad de operaciones de lectura por segundo que una tabla puede manejar. Cada unidad de lectura aprovisionada corresponde a una operación de lectura de hasta 4 KB de datos.
- **write_capacity** = 1. Al igual que en las unidades de lectura, las unidades de escritura aprovisionadas determinan la capacidad de escritura de una tabla. Cada unidad de escritura aprovisionada corresponde a una operación de escritura de hasta 1 KB de datos.

Además, establecemos las claves para cada una de las tablas, siguiendo el formato ID_nombre_tabla, y siendo todas de tipo numérico:

- ID_Electrolinera.
- ID_PuntoCarga.
- ID_Estado.
- ID_Estadisticas.

A.2 Backups periódicos en S3

Amazon S3 (*Simple Storage Service*) es un servicio de almacenamiento en la nube altamente escalable y duradero. Está diseñado para ofrecer una capacidad de almacenamiento casi ilimitada y de alta disponibilidad. Los datos se almacenan en unidades lógicas llamadas *buckets*, que permiten organizar la información y configurar las políticas de acceso. En este trabajo, se utiliza S3 para el almacenamiento automático y programado de las copias de seguridad de la información contenida en las tablas de DynamoDB. Para ello, tenemos que seleccionar la tabla de la que hacer copias de seguridad, la política de las copias de seguridad, el bucket de destino y los permisos y roles necesarios.

También sería posible establecer un PITR (*Point in Time Recovery*), lo cual nos permitiría recuperar

el estado de la tabla en un punto determinado del pasado, pero preferimos el almacenamiento de información en otro servicio de AWS.

A.3 Actualización del estado

Una parte importante de la arquitectura del sistema es la actualización del estado de los diferentes puntos de carga instalados. Se realiza a través del uso de una Raspberry Pi, la cual tiene conectados dos periféricos: un sensor de ultrasonidos y una cámara.

Estos sensores permiten monitorizar el estado del punto de carga y a través de una petición HTTP a una Lambda de AWS, actualizará la base de datos correspondiente al estado de los puntos de carga.

La actualización del estado será la primera parte invocada en el sistema, pues será utilizado por el resto del sistema para realizar la consulta de datos. Esta parte de la arquitectura será detallada en la Sección II-D.

A.4 Consulta de datos

El módulo de consulta de datos está, a su vez, compuesto de dos submódulos:

1. **Web de control y estadísticas**. Este submódulo proporcionará un frontend mediante el cual los usuarios puedan consultar la ubicación y estado de las electrolineras. En la Sección II-B, se desarrollará el funcionamiento de este submódulo.
2. **Car API**. Este submódulo proporcionará una interfaz tipo *REST API* mediante la que el software que se ejecute en los sistemas embarcados pueda consultar la ubicación y estado de las electrolineras. La Sección II-C ofrece más información al respecto de este submódulo.

Además, en este módulo se integrará el servicio de autenticación mediante Cognito, lo que permitirá que solo usuarios autenticados hagan uso del sistema.

B. Página web de control y estadísticas

Para alojar la página web de control y estadísticas, hemos optado por un servidor virtual en EC2 (*Elastic Computing Cloud*) dentro de Amazon Web Services.

EC2 permite disponer de servidores virtuales en la nube, con capacidad de escalar vertical y horizontalmente según la demanda o las necesidades de la aplicación. Dispone de diferentes familias de máquinas virtuales, cada una destinada a casos de uso diferentes. Junto con EC2 también es posible configurar la red subyacente de forma avanzada, a través de VPC (*Virtual Private Cloud*), permitiendo definir diferentes redes, tablas de enrutamiento personalizadas y control del tráfico entrante y saliente. Finalmente, servicios como ELB (*Elastic Load Balancer*) y Auto-Scaling permiten la construcción de arquitecturas altamente disponibles y escalables.

En este trabajo se ha seleccionado una instancia de la familia **t2**, más concretamente la **t2.micro**, que dispone de 1 vCPU, 1 GiB de memoria RAM y un almacenamiento en bloques de 30 GiB. La imagen seleccionada ha sido la Amazon Linux 2023. Junto con la máquina se ha desplegado una VPC, compuesta por:

- Una **subred privada**, para que las máquinas virtuales puedan comunicarse entre ellas de manera segura.
- Una **subred pública**, para emplazar las máquinas expuestas a internet.
- Una **NAT gateway**, que permitirá la comunicación entre la red pública y la privada, para permitir que las máquinas de la red privada accedan a Internet.
- Una **internet gateway**, que permite el acceso de Internet a la red pública.
- Una **dirección IP elástica**, que será asignada a la instancia EC2.
- Dos **tablas de enrutamiento** para la red pública y privada.
- Dos **rutas estáticas**, que definan cómo salir a Internet en la red pública y privada.
- Dos **asociaciones de tablas de enrutamiento con subredes**, para asociar las tablas con la subred correspondiente.
- Un **grupo de seguridad**, que permitirá definir el tráfico entrante y saliente permitido.

En la Figura 2 se observa un diagrama de ejemplo de la configuración de red usada.

Haciendo uso del campo *user_data* cuando desplegamos la instancia de EC2 realizamos las tareas previas e instalamos las dependencias necesarias. Son las siguientes:

1. Actualización de los paquetes disponibles en el repositorio.
2. Instalación de Git, Node.js, Python y pip.
3. Clonación del repositorio de GitHub localmente.
4. Importación de datos iniciales a la base de datos con Python.
5. Instalación del servidor web PM2.
6. Instalación de dependencias del servidor web.
7. Puesta en marca del servidor web.
8. Configuración de la tarea cron que genera estadísticas cada 30 segundos.

La aplicación web ha sido creada usando Node.js, un entorno de ejecución de Javascript del lado del servidor. Node.js ofrece un rendimiento eficiente gracias a su programación asíncrona y basada en eventos. Dispone de un amplio ecosistema de paquetes y bibliotecas que ofrecen una gran flexibilidad. Para acceder a la base de datos datos, se usará el módulo de DynamoDB de la librería de AWS para Node.js.

Finalmente, para iniciar la aplicación web en un entorno de ejecución se usa PM2 (*Process Monitor 2*). PM2 permite administrar y supervisar de forma eficiente aplicaciones. Dispone de características avanzadas como el balanceo de carga o la administración de clústeres. Su monitor registra el rendimiento de las aplicaciones, como el consumo de CPU y

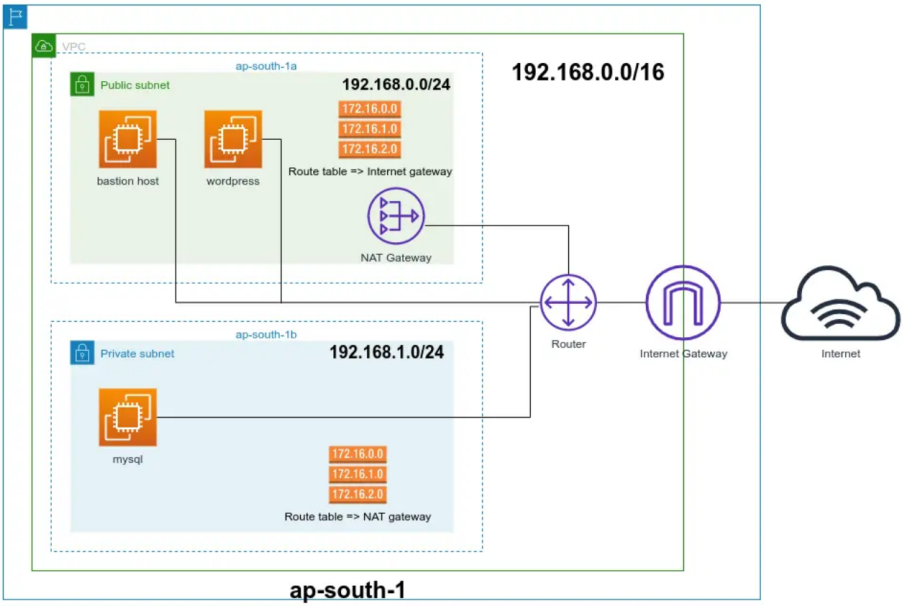

Fig. 2: Diagrama de ejemplo de la configuración de red usada. Los servicios y las direcciones de red no se corresponden con este proyecto.

memoria, tiempo de respuesta y errores. Permite establecer políticas de reinicio cuando una aplicación presenta un malfuncionamiento, así como la actualización sin tiempo de inactividad de las aplicaciones.

A través de la aplicación web disponemos de varias vistas:

- **Vista principal**: Muestra el número y la localización de las electrolineras y en número de puntos de carga en un mapa. Un ejemplo es la Figura 3.
- **Vista de electrolinera**: Muestra el nombre, estado, ubicación y puntos de carga de una electrolinera. Un ejemplo es la Figura 4.
- **Vista de punto de carga**: Muestra el número, estado actual y estados pasados de un punto de carga de una electrolinera. Un ejemplo es la Figura 5.
- **Vista de estadísticas**: Muestra un gráfico donde podemos ver cómo ha evolucionado el número de puntos de carga sin uso, en uso y fuera de servicio a lo largo del tiempo. Un ejemplo es la Figura 6.

Dado que el servicio EC2 no incluye técnicas de auto-escalado implícitas (sin que el usuario lo configure), para disfrutar de alta disponibilidad, se debería activar el servicio EC2 Auto Scaling. De esta forma, ante la falta de recursos por las instancias en ejecución, nuevas instancias serían ejecutadas, escalando horizontalmente. Para dividir la carga entre el conjunto de instancias en ejecución, también se puede configurar un ELB (*Elastic Load Balancer*), que irá repartiendo las peticiones entrantes entre el conjunto de instancias. Ambas configuraciones, tanto el auto escalado, como el balanceo de carga, no han sido configurados en el despliegue del sistema actual.

C. *API HTTP para sistemas ciberfísicos en vehículos eléctricos*

El objetivo de este segundo módulo de la arquitectura es el de ofrecer una API sencilla para que los programadores de los sistemas ciberfísicos instalados en los vehículos eléctricos puedan implementar llamadas a la misma. De esta forma, podrían crear una aplicación funcional en el propio vehículo que comprobase el estado y ubicación de diversas electrolineras y/o puntos de carga de las mismas para, por ejemplo, mostrarlo en la pantalla incluida en el propio vehículo, o calcular cuál es el punto de carga

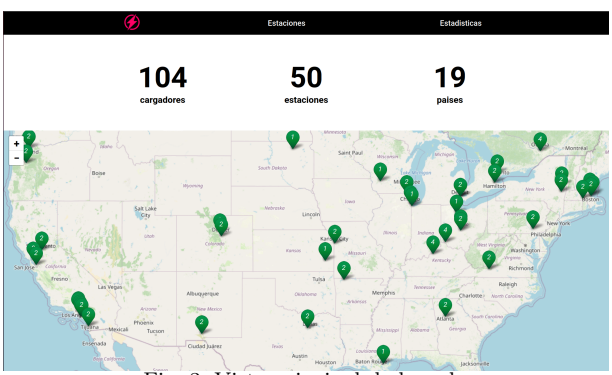

Fig. 3: Vista principal de la web.

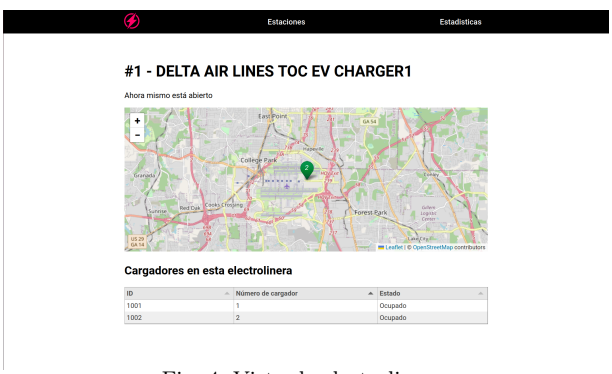

Fig. 4: Vista de electrolinera.

cercano más óptimo en función de métricas como la distancia al mismo, la autonomía actual del vehículo, la ubicación de la electrolinera, su nivel de ocupación, estado, etc.

C.1 Función principal.

Principalmente, la API ofrecerá la función de obtener todas las electrolineras ubicadas dentro de un área rectangular, así como sus puntos de carga asociados y sus correspondientes estados. Así, el sistema embarcado en el vehículo llamaría a la API mediante una petición HTTP POST enviándole 4 parámetros:

- **lat1**. Latitud de la primera esquina del rectángulo.
- **lat2**. Latitud de la segunda esquina del rectángulo.
- **lon1**. Longitud de la primera esquina del rectángulo.
- **lon2**. Longitud de la segunda esquina del rectángulo.

Estos cuatro parámetros, que formarían 2 puntos en un mapa, determinan las 2 esquinas del área rectangular mostrada en el mapa de la pantalla del vehículo.

C.2 Otras funciones.

Adicionalmente, la API también ofrecerá otras 2 funciones:

1. **Reserva de punto de carga**. Si recibe un parámetro que contenga la matrícula de un coche y el identificador de un punto de carga, podrá reservar ese punto de carga, marcándolo como ocupado, si es que se encuentra libre en ese determinado momento.
2. **Consulta del estado de un solo punto**. Si la API recibe el identificador de un punto de carga existente, devolverá el estado de ese único punto de carga al coche.

Fig. 5: Vista de puntos de carga y sus estados.

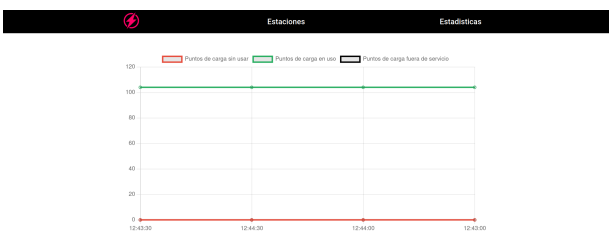

Fig. 6: Vista de estadísticas de uso.

C.3 Detalles de implementación

Dado que la API tendrá un funcionamiento muy sencillo, se ha optado por implementarla mediante una función Lambda. De esta manera, no será necesario preocuparse por el hardware sobre el que se ejecute la función, y tendremos el escalado garantizado, ya que Lambda nos proporciona autoescalado. Además, el coste de la función no debería ser muy elevado, dado que solo pagaremos por cada invocación a la función. Sin embargo, hay que considerar que, en caso de que el sistema escalase masivamente, esta podría no ser la mejor opción desde el punto de vista económico. Finalmente, otra ventaja de Lambda es que reducirá el tiempo de desarrollo, ya que solo será necesario preocuparse del código de la propia función, y no de factores externos.

Respecto a la implementación de la función, se ha realizado en el lenguaje de programación Python 3, dado que se trata de un lenguaje soportado por Lambda y que es sencillo de leer y escribir, facilitando un desarrollo y depuración de errores rápidos. Para comunicar la función con los servicios de AWS, se ha utilizado el SDK de AWS para Python, Boto3.

C.4 Servicios de AWS implicados.

En cuanto a los servicios de AWS con los que se comunicará la función Lambda, encontramos 2: *DynamoDB* y *Cognito*. A continuación, se detalla la interacción de la función con ambos componentes:

1. **DynamoDB**. DynamoDB es el servicio utilizado para el almacenamiento de toda la información implicada en la aplicación. Por lo tanto, la función Lambda se comunicará con las tablas *Estado* y *Electrolinera* para consultar la información que será proporcionada al vehículo que haya realizado una llamada a la API.
2. **Cognito**. Cognito es el servicio de autenticación y control de acceso de AWS. En este caso, servirá para que solo los usuarios (vehículos) autenticados puedan realizar llamadas a la función, reduciendo, así, el riesgo de ataques con una cantidad masiva de peticiones que puedan disparar el coste de la infraestructura. De esta manera, se añade una capa de seguridad al sistema, también de manera escalable (dado que Cognito es capaz de escalar hasta millones de usuarios [1]).

C.5 Prototipo de cliente.

Finalmente, para comprobar el funcionamiento de la API, se proporcionará un prototipo de cliente muy sencillo, también en Python, que hará llamadas a la API y mostrará su respuesta a través de la terminal. Este prototipo da una idea de implementación muy

básica que podría servir como punto de partida para desarrollar una aplicación más compleja que realice llamadas al servicio propuesto.

D. Sensores y estado de los puntos de carga

Este tercer módulo tiene como objetivo principal actualizar el estado de los diferentes puntos de carga del sistema, integrados en las diferentes estaciones de servicio. Estos puntos de carga podrán tener tres diferentes estados, declarados como:

- **0:** Punto de carga libre, listo para ser utilizado.
- **1:** Punto de carga ocupado, espere para poder utilizarlo.
- **2:** Punto de carga averiado o en reparaciones.

Para poder actualizar estos estados, en cada punto de carga se instalaría una placa Raspberry Pi 3B [2], o de cualquier otra familia, un sensor de ultrasonidos HC-SR04[3] y una cámara V2 Raspberry Pi Sony 8MPX [4].

El sensor HC-SR04 es un sensor de distancia de bajo costo que utiliza ultrasonido para determinar la distancia de un objeto en un rango de 2 a 450 cm. Destaca por su pequeño tamaño, bajo consumo energético, buena precisión y excelente precio. El funcionamiento del sensor es el siguiente: el emisor piezoeléctrico emite 8 pulsos de ultrasonido(40KHz) luego de recibir la orden en el pin TRIG, las ondas de sonido viajan en el aire y rebotan al encontrar un objeto, el sonido de rebote es detectado por el receptor piezoeléctrico, luego el pin ECHO cambia a Alto (5V) por un tiempo igual al que demoró la onda desde que fue emitida hasta que fue detectada, el tiempo del pulso ECHO es medido por el microcontrolador y así se puede calcular la distancia al objeto. El funcionamiento del sensor no se ve afectado por la luz solar o material de color negro.

Estos sensores detectarán la presencia de un objeto a menos de $0,5$ m durante 10 segundos. Estos 10 segundos permitirán al sensor asegurarse que no es un objeto casual, como una hoja, que puede acercarse al sensor. Si durante 10 segundos de forma continua el sensor se encuentra ocupado, este efectuará el envío de una petición al sistema para actualizar el estado del punto de carga. De esta misma forma, cuando tras estar ocupado el sensor no detecte ningún objeto, volverá a actualizar el estado del punto de carga liberándolo.

La cámara V2 Raspberry Pi Sony 8MPX se conecta directamente al conector CSI de cualquier Raspberry Pi con la última versión de Raspbian. Ofrece una resolución de 8 megapíxeles (3280x2464) en imagen y 1080p en grabación de vídeo a 30fps, 720p a 60fps y 480p a 90fps. La cámara está fabricada con un sensor Sony Exmor R IMX219 que permite unas imágenes claras con un foco fijo y de muy alta calidad.

Esta cámara permitirá la detección de matrículas en los vehículos, utilizada en el envío de la información para poder realizar estadísticas de los vehículos que utilizan las instalaciones.

De esta forma, la información enviada por el punto de carga será:

- **%ID_PuntoCarga%.%Timestamp%:** Key del item que incluye el ID sobre el que se encuentra instalada la placa Raspberry y el momento en el que se registra un cambio de estado del punto de carga.
- **Estado:** Estado del punto de carga, representado por los valores numéricos expresados previamente.
- **Matrícula:** Matrícula del vehículo que interacciona con el punto de carga.
- **Tiempo:** Momento en el que se registra el cambio de estado en el punto de carga.

Toda la información de los sensores será enviada a través de una petición HTTP Post de la Raspberry Pi y recibida por una función Lambda en el servicio ofrecido por AWS. Esta función se comunicará a través del SDK Boto3 con la tabla *Estado* de *DynamoDB*. De esta forma, recogiendo la información contenida en el campo *event*, se realizará una petición a DynamoDB para crear un nuevo item en dicha tabla.

Tras todo este proceso, si la comunicación ha sido positiva, se enviará un mensaje de estado 200, que representa que toda la comunicación se ha realizado de forma positiva y la tabla ha sido actualizada.

III. Despliegue

Terraform es una herramienta de código abierto que permite gestionar y desplegar infraestructuras de manera automatizada. Se basa en el concepto de infraestructura como código (IaC), lo que significa que la infraestructura se define mediante archivos de configuración en lugar de configurarse manualmente.

Una de las principales ventajas de utilizar Terraform es su enfoque declarativo. En lugar de describir los pasos para llegar a un estado deseado, se define el estado deseado directamente. Esto facilita la comprensión y el mantenimiento del código, ya que se enfoca en lo que se quiere lograr en lugar de cómo hacerlo.

Otra ventaja de Terraform es su compatibilidad con múltiples proveedores de nube, incluyendo AWS. Esto significa que se puede utilizar el mismo conjunto de archivos de configuración para desplegar infraestructuras en diferentes entornos, lo que proporciona coherencia y flexibilidad.

Al utilizar Terraform con AWS, se obtiene una serie de beneficios adicionales. Terraform proporciona una capa de abstracción sobre los servicios de AWS, lo que facilita la gestión y el despliegue de recursos sin tener que lidiar con los detalles subyacentes de la infraestructura. Además, permite mantener un control total sobre los recursos y garantizar la reproducibilidad del entorno.

Terraform también es altamente escalable y se integra con otros servicios de AWS, como IAM, lo que facilita la definición de políticas de acceso y la gestión de permisos de forma centralizada.

Debido a las ventajas ofrecidas por Terraform, será la herramienta que utilizaremos para desplegar de forma automatizada nuestro sistema. Para ello hemos utilizado una estructura en la cual un único fichero *main.tf* será el encargado de invocar al resto de configuraciones. La estructura en diferentes módulos nos permitirá manejar de forma aislada cada uno de los principales componentes, los cuales serán:

- **DynamoDB:** Será el encargado de crear la base de datos, formada por las 4 tablas (mirar Sección II-A.1) con las keys correspondientes a cada una de ellas. El recurso empleado para crear las tablas será *aws_dynamodb_table*[5].
- **Lambdas:** Será el encargado de implementar y desplegar las funciones de Lambda. Utilizaremos el recurso *aws_lambda_function* de Terraform para crear y configurar cada una de las funciones[6]. Además, se utilizará el recurso *aws_lambda_function_url* para exponer la URL de la lambda y poder invocarla desde otros hosts[7].
- **EC2:** Será el encargado de crear y administrar las instancias de EC2 necesarias para ejecutar los servicios y aplicaciones requeridos. Utilizaremos el recurso *aws_instance* de Terraform para definir y configurar las instancias de EC2[8]. Podremos especificar el tipo de instancia, la imagen AMI a utilizar, las opciones de red, almacenamiento y otras configuraciones necesarias para cada instancia. Esto nos permitirá desplegar y gestionar de manera automatizada las instancias de EC2 necesarias para nuestra infraestructura. Además, será necesario crear recursos como *aws_vpc, aws_gateways, aws_subnets* y *aws_route_tables* entre otros.

IV. Pruebas de funcionamiento

A. Pruebas de carga al servidor web

A.1 Carga promedio

Esta prueba de carga consiste en realizar peticiones a la web de forma constante y prolongada, asumiendo que, en periodo normal, la web debe responder entre 3 y 5 peticiones por segundo. Por tanto, se ha establecido la carga en un total de 250 peticiones por minuto. La prueba ha tenido una duración de 1 minuto y su resultado puede verse en la Figura 7. Se observa como el tiempo de respuesta medio se mantiene en 40 ms, y existe poca variación entre los tiempos de respuesta. El tiempo de respuesta mínimo ha sido de 25 ms, mientras que el máximo ha sido 125 ms.

A.2 Carga aumentada de forma prolongada

Esta prueba de carga consiste en ir aumentando de forma constante el número de peticiones entrantes, para conocer el comportamiento del despliegue ante una situación donde cada vez más usuarios están usando la web.

Para ello, se ha establecido la carga de la prueba de 0 a 300 usuarios por segundo, aumentando progresivamente, con una duración de 1 minuto. Su resultado se observa en la Figura 8. Se puede ver cómo el despliegue realizado no consigue manejar tal número de peticiones, y que, a partir de 60 pet./seg. el tiempo de respuesta aumenta considerablemente. La media se establece en 3.7 seg., con un máximo de 10 seg. para responder a una petición.

Después de analizar este bajo rendimiento, descubrimos que DynamoDB estaba suponiendo un cuello de botella. Más en especifico, las unidades de lectura aprovisionadas no eran suficientes para atender este tipo de cargas. Por ello, tras aumentar el número de unidades a 200, repetimos la prueba, dando lugar a la Figura 9. En este caso, se puede observar cómo los tiempos de respuesta son mucho más estables, donde el tiempo crece de forma proporcional al número de usuarios, establecido en alrededor de 600 ms el tiempo de respuesta medio, y el tiempo máximo en 1.2 seg.

A.3 Carga pico

Finalmente, y para concluir con las pruebas de carga en la web, hemos querido estudiar el comportamiento del despliegue cuando cientos de peticiones llegan de forma inmediata. Para ello, hemos establecido la carga en torno a 300 peticiones por segundo, 15.000 por minuto en total. La configuración de las tablas se ha establecido en las 200 unidades de lectura comentadas anteriormente.

En al Figura 10 se observan los resultados. Se ha obtenido una media de 400 ms en el tiempo de respuesta, con una máxima de 1.2 seg. Podemos ver como, durante la prueba, se ha mantenido un tiempo de respuesta razonablemente estable, salvo al inicio, donde más de 500 peticiones por segundo han llegado de forma inmediata.

V. Conclusiones

En este trabajo se presenta una aplicación basada en cloud y edge computing, que permite la monitorización y la consulta del estado de ocupación de puntos de carga para vehículos eléctricos. Se explica cómo se ha realizado el despliegue y la implementación de los diferentes componentes del sistema, así cómo las consideraciones llevadas a cabo en materia de escalabilidad y alta disponibilidad. También se indican algunas consideraciones de diseño llevadas a cabo. Para finalizar, se presentan algunas pruebas de carga realizadas.

En general, se observa como el despliegue realizado cumple el objetivo propuesto, ofreciendo un nivel de rendimiento robusto y presentando un prototipo totalmente funcional.

Referencias

[1] "AWS — Gestión de identidades y autenticación de usuario en la nube — aws.amazon.com," `https://aws.amazon.com/es/cognito/`, [Accessed 15-May-2023].
[2] Ltd, "Buy a Raspberry Pi 3 Model B – Raspberry Pi," May 2023, [Online; accessed 15. May 2023].
[3] "Sensor Ultrasonido HC-SR04," May 2023, [Online; accessed 15. May 2023].
[4] "Cámara v2 Raspberry Pi Sony 8Mpx," May 2023, [Online; accessed 15. May 2023].

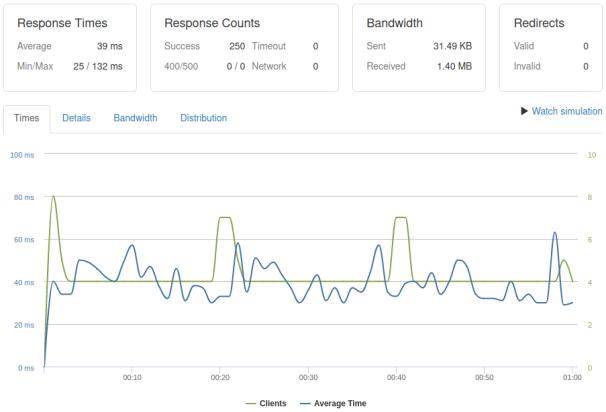

Fig. 7: Prueba de carga sobre la web con carga promedia estimada

Fig. 8: Prueba de carga rampa de 0 a 300 usuarios con 1 unidad de lectura en las tablas.

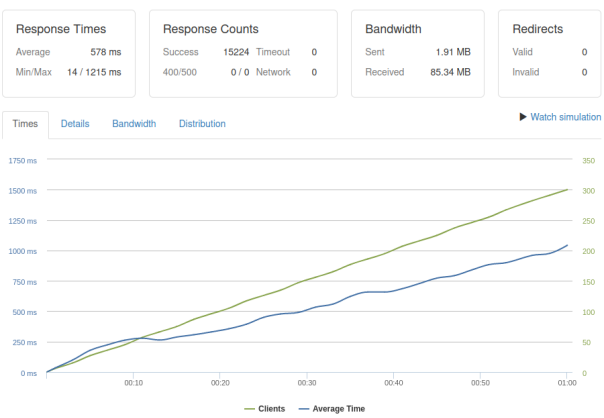

Fig. 9: Prueba de carga rampa de 0 a 300 usuarios con 200 unidad de lectura en las tablas.

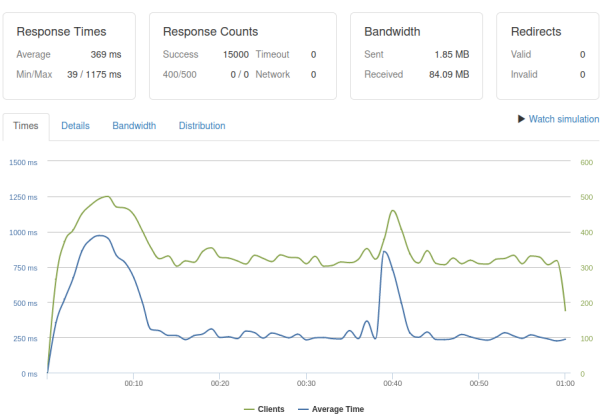

Fig. 10: Prueba de carga pico de 300 usuarios por segundo.

[5] "aws_dynamodb_table | Resources | hashicorp/aws | Terraform Registry," May 2023, [Online; accessed 16. May 2023].

[6] "aws_lambda_function | Resources | hashicorp/aws | Te-

rraform Registry," May 2023, [Online; accessed 16. May 2023].
[7] "aws_lambda_function_url | Resources | hashicorp/aws | Terraform Registry," May 2023, [Online; accessed 16. May 2023].
[8] "aws_instance | Resources | hashicorp/aws | Terraform Registry," May 2023, [Online; accessed 16. May 2023].