
# Aplicación de tecnologías y arquitecturas serverless para el desarrollo de soluciones IoT

**Eduardo Sanfrutos**
Máster Universitario en Ingeniería de Telecomunicación
Telemática

**Jose Lopez Vicario**
**Xavi Vilajosana Guillen**

junio, 2020

**FICHA DEL TRABAJO FINAL**

| | |
|---|---|
| **Título del trabajo:** | *Aplicación de tecnologías y arquitecturas serverless para el desarrollo de soluciones IoT* |
| **Nombre del autor:** | *Eduardo Sanfrutos Ruiz* |
| **Nombre del consultor/a:** | *Jose Lopez Vicario* |
| **Nombre del PRA:** | *Xavi Vilajosana Guillen* |
| **Fecha de entrega (mm/aaaa):** | 06/2020 |
| **Titulación::** | *Máster Universitario en Ingeniería de Telecomunicación* |
| **Área del Trabajo Final:** | *Telemática* |
| **Idioma del trabajo:** | *Castellano* |
| **Palabras clave** | • *IoT*<br>• *FaaS*<br>• *Cloud* |

**Resumen del Trabajo (máximo 250 palabras):**


El proyecto estará orientado a investigar el concepto de arquitectura *serverless* o *FaaS* para soluciones *IoT*. Debido a la disrupción de las tecnologías y su gran velocidad de desarrollo aparece una necesidad de poder implementar soluciones escalables, flexibles, estables y con costes bajos para las soluciones *IoT*. Por lo tanto, el concepto de arquitectura *serverless* encaja perfectamente con estas necesidades. Pudiendo conseguir desplegar soluciones *IoT* sin la necesidad de tener una infraestructura, ni costes de mantenimiento y prácticamente ningún coste de operación.

Inicialmente se hará una investigación de las tecnologías Cloud que permites realizar arquitecturas *serverless* para soluciones *IoT* y su estado del arte. Entre los que aparecen los conceptos *IaaS, PaaS, CaaS y FaaS.* Este último concepto será el más importante ya que es la tecnología que permitirá establecer una arquitectura *serverless IoT*. Esto se debe a las propias características de *FaaS* que son contenedores sin estado que se ejecutan bajo eventos, escalables y pueden ser desplegados independientemente de la infraestructura subyacente.

Una vez se han estudiado los conceptos de *serverless* y *FaaS*, además de sus diferencias y similitudes, se implementará una solución Cloud para validar lo expuesto. En concreto una implementación de un regulador de temperatura para *Smart Home*. Esta solución tendrá uno nodo *edge* con *FaaS* que se encargará del control y computación de los elementos *IoT*. Finalmente se sacarán conclusiones de este tipo de arquitectura para soluciones IoT para comprobar a nivel de costes, escalabilidad, mantenimiento y *time-to-market.*

**Abstract (in English, 250 words or less):**

This project investigates the concept of serverless architecture or FaaS for IoT solutions. Due to the disruption of technologies and their high speed of development there is a need for innovative, stable and low-cost solutions for IoT. Therefore, the concept of serverless architecture fits perfectly with the requirements for deploying IoT solutions without the need of infrastructure, maintenance costs and almost no operating costs.

Firstly, there will be a investigation of Cloud technologies that allowed serverless architectures for IoT solutions and their state of the art. Among it the concepts of IaaS, PaaS, CaaS and FaaS appear. The final concept is the most important because it is the technology that allows to establish a serverless IoT architecture due to it *serverless* features, which are stateless containers that run under events, scalable, and they can be deployed regardless of the underlying infrastructure.

Once the concepts of a serverless architecture and FaaS has been studied, including its difference and similarities, a Cloud solution will be implemented to validate what has been presented. Specifically, a temperature regulator for Smart Home was used. The result is an edge node with FaaS that will be in charge of controlling and computing the IoT elements. Finally, a conclusion will be established about this type of architecture, specifically for IoT solutions verifying cost, scalability, maintenance and time-to-market.

# Índice

## Tabla de contenido

## Lista de figuras

# 1 Introducción

## 1.1 Contexto y justificación del Trabajo

El proyecto estará orientado a investigar el concepto de arquitectura *serverless* o *FaaS* (*Functions as a Service*) para soluciones *IoT*. Debido a la disrupción de las tecnologías y su gran velocidad de desarrollo aparece una necesidad de poder implementar soluciones escalables, flexibles, estables y con costes bajos para las soluciones *IoT*. Por lo tanto, el concepto de arquitectura *serverless* encaja perfectamente con estas necesidades. Pudiendo conseguir desplegar soluciones *IoT* sin la necesidad de tener una infraestructura, ni costes de mantenimiento y prácticamente ningún coste de operación.

Inicialmente se hará una investigación de las tecnologías Cloud que permites realizar arquitecturas *serverless* para soluciones *IoT* y su estado del arte. Entre los que aparecen los conceptos *IaaS (Infrastructure as a Service), PaaS (Platform as a Service), CaaS (Container as a Service) y FaaS (Function as a Service).* Este último concepto será el más importante ya que es la tecnología que permitirá establecer una arquitectura *serverless IoT*. Esto se debe a las propias características de *FaaS* que son contenedores sin estado que se ejecutan bajo eventos, escalables y pueden ser desplegados independientemente de la infraestructura subyacente.

Una vez se ha estudiado los conceptos de *serverless* y *FaaS*, además de sus diferencias y similitudes, se implementará una solución Cloud para validar lo expuesto, en concreto una implementación de un regulador de temperatura para *Smart Home*. Esta solución tendrá uno nodo *edge* con *FaaS* que se encargará del control y computación de los elementos *IoT*. Finalmente se sacarán conclusiones de este tipo de arquitectura para soluciones IoT para comprobar a nivel de costes, escalabilidad, mantenimiento y *time-to-market*.

## 1.2 Objetivos del Trabajo

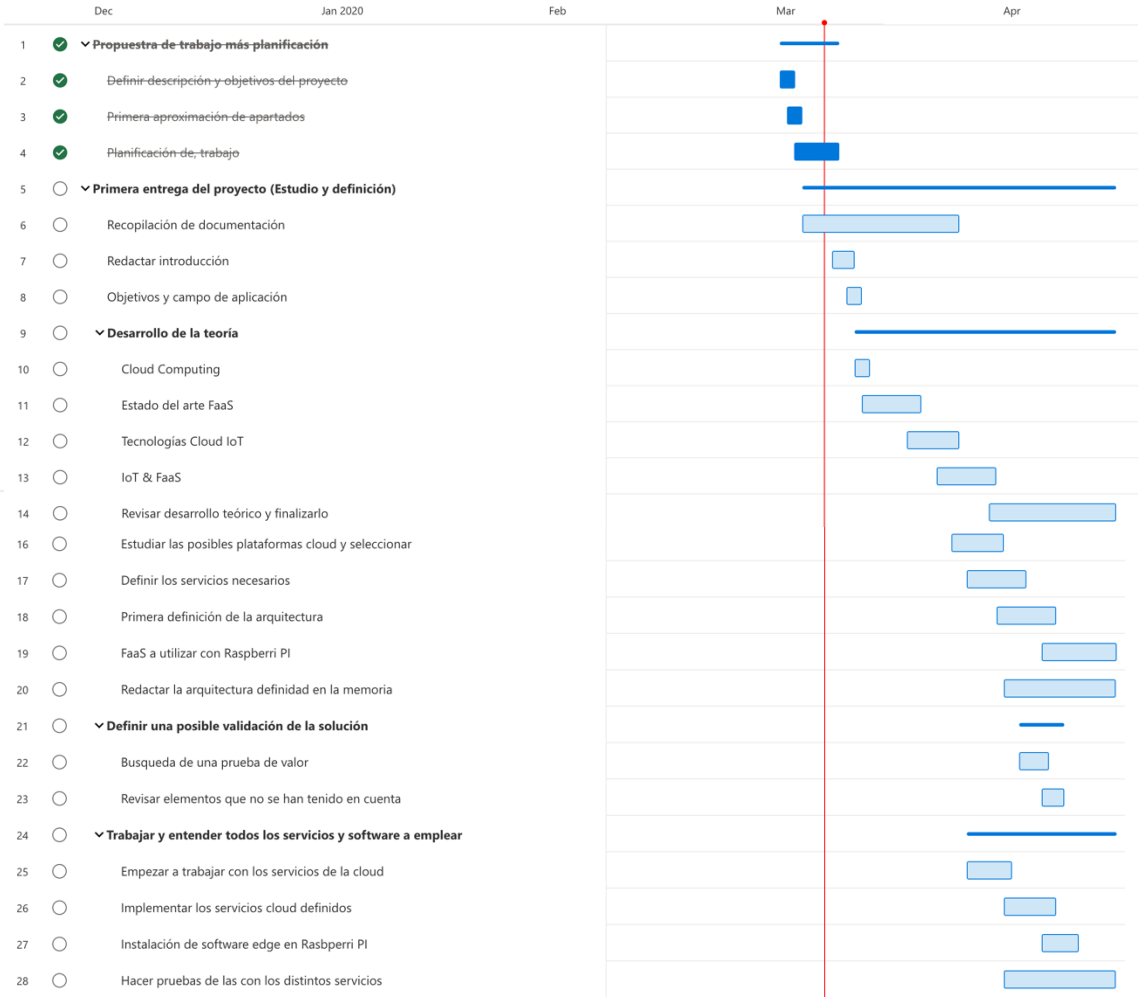
- Investigar los modelos *Cloud.*
- Investigar las posibles arquitecturas *IoT.*
- Investigar las arquitecturas *serverless* para soluciones *IoT.*
- Implementar un caso de uso real con *serverless IoT*.
- Validar la solución.

## 1.3 Enfoque y método seguido

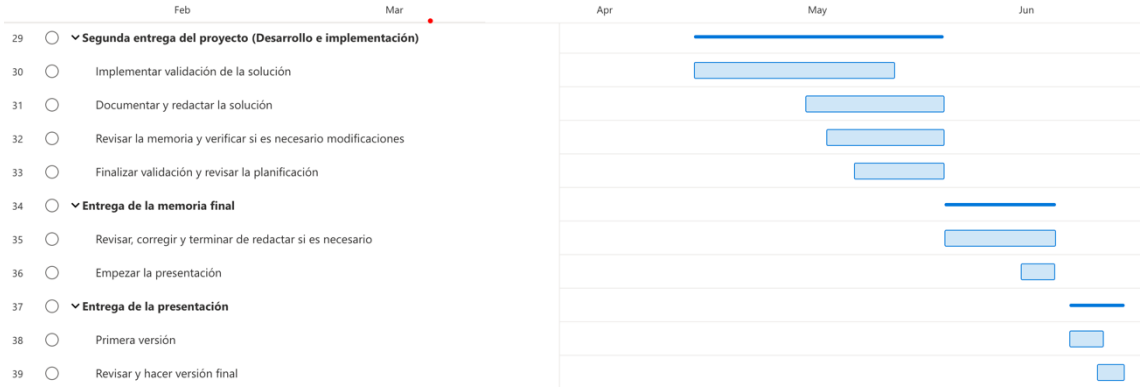
La estrategia seguida para la elaboración del proyecto ha sido ir explorando las tecnologías que permiten establecer una solución que encaje con el concepto de *serverless IoT*. Una vez se han analizado, se ha trabajado con una tecnología que permitiese poner en práctica lo expuesto, con lo que se ha conseguido adaptar un producto existente de *Smart Home* a esta tecnología. Está es la mejor estrategia debido a el carácter innovador que tiene este proyecto, donde se partía de una base meramente teórica y se ha acabado pudiendo exponerlo desarrollando un caso de uso real mediante esta tecnología.

## 1.4 Planificación del Trabajo

### 1.4.1 Primera parte



### 1.4.2 Segunda parte y final




Si hay que tener en cuenta todas las complicaciones que han ido ocurriendo, la finalización del proyecto se ha retrasado una semana más de lo previsto por lo que la planificación no ha sido del todo correcta. Esto se deba a que no se ha tenido en cuenta las posibles complicaciones que interrumpan el desarrollo continuo del trabajo. Por suerte, se ha podido seguir los pasos que han permitido encontrar una tecnología adecuada y llegar a una solución óptima.

También, hay que valorar que se partía de la base de desconocimiento absoluto de las posibles tecnologías que podían permitir realizar un despliegue *serverless IoT*, con lo que parte del trabajo ha sido encontrar una que encajase con las características deseadas para validar la solución

## 1.5 Breve sumario de productos obtenidos

Se ha simulado una aplicación de un regulador de temperatura que, mediante dispositivos de calefacción y climatización, permita ajustar la temperatura en base a una temperatura de referencia. Todo ello mediante dispositivos como servicio, utilizando los GPIO para poder observa una interacción en tiempo real, simulando el calefactor y climatizador con un par de leds.

## 1.6 Breve descripción de los otros capítulos de la memoria

El proyecto comienza con una parte más de investigación, estado del arte y descripción tecnológica de tres elementos clave. El primero es Cloud *Computing*, el segundo que depende del primero es la arquitectura *serverless* y, por último, las tecnologías *IoT*. Los tres casos tienen una distribución similar de definición, características, modelo o arquitectura y de ventajas y desventajas. Para acabar se ha hecho un breve resumen de lo visto en ese apartado.

El siguiente punto ha sido un análisis de lo que puede aportar una arquitectura *serverless* para una solución *IoT*. Una vez analizado, en el siguiente apartado se ha hecho una propuesta de la solución que, finalmente, permita valorar la solución según lo que se ha ido analizando en el penúltimo apartado. Por último, se ha realizado una conclusión que resuma todos los puntos explorados en el proyecto.

# 2 Estado del arte y descripción tecnológica

## 2.1 *Cloud Computing*

El origen de *Cloud Computing* aparece por la necesidad de reducir la inversión en recursos de computación propios (*on premise)*. Estos recursos de computación han de tener una planificación previa en cuanto a su dimensionamiento. Por ello, surge la posibilidad de que los recursos de computación estén infrautilizados y, además, al ser recursos limitados por su dimensionamiento no es posible anteponerse a las necesidades de escalabilidad o picos de computación que puedan aparecer.

Todo esto implica una gran inversión de capital (*CapEx*) [1] además de todos los costes implicados para el mantenimiento y gestión de estos recursos de computación. Si, en vez tener que invertir de antemano en los recursos de computación, se pudiese pagar únicamente por lo que se necesita implicaría solo una inversión de operación (*OpEx*). Aquí es donde aparece el concepto de *Cloud Computing* que permite utilizar un modelo de pago por uso o *pay as you use*.

Según la definición del *National Insitute of Standards and Technology* (*NIST*):

> "[4] La computación en la nube es un modelo para permitir el acceso de red ubicuo, conveniente y bajo demanda a un grupo compartido de recursos de computación configurables (por ejemplo, redes, servidores, almacenamiento, aplicaciones y servicios) que pueden aprovisionarse y liberarse rápidamente con mínimo esfuerzo de gestión o interacción por parte del proveedor de servicios. Este modelo de nube promueve la disponibilidad y se compone de cinco características esenciales, tres modelos de servicio y cuatro modelos de implementación."

Está definición es del 2011 por lo que para el caso de los modelos de servicio han ido apareciendo modelos adicionales que ser verán más adelante. Aún así, es la definición más exacta y menos ambigua que puede haber, por lo tanto, para este documento esta se usará como la definición de referencia.

### 2.1.1 Modelos de despliegue *Cloud*

En este apartado se va a explicar brevemente los distintos modelos de despliegue *Cloud* y sus características. [4]

#### 2.1.1.1 Privada

La infraestructura *Cloud* está provisionada exclusivamente para el uso por parte de una única organización con múltiples consumidores. Puede pertenecer, estar gestionada, y operada por la organización, un tercero, o una combinación de ambos, y puede ser *on* o *off premise.*

#### 2.1.1.2 Comunitaria

La infraestructura *Cloud* está provisionada para el uso exclusivo de una comunidad específica de consumidores de organizaciones con mismos intereses. Puede pertenecer, estar gestionada, y operada por una o varias organizaciones de la comunidad, un tercero, o una combinación de ambos, y puede ser *on* o *off premise.*

#### 2.1.1.3 Pública

La infraestructura *Cloud* está provisionada para el uso abierto del público general. Puede pertenecer, estar gestionada, y operada por una empresa, organización gubernamental o académica, o una combinación de ambos. Es *on premise* del proveedor *Cloud.*

#### 2.1.1.4 Hibrida

La infraestructura *Cloud* es una combinación de dos o más infraestructuras *Cloud* distintas (privada, comunitaria o pública) que permanecen como entidades únicas, pero que están enlazadas entre ellas por tecnología estandarizada o propietaria que permite la portabilidad de datos y aplicaciones.

### 2.1.2 Características

En este apartado se definen las características principales que definen *Cloud Computing* fundamentales para poder entender mejor el concepto. [4]

#### 2.1.2.1 Auto servicio bajo demanda:

Un consumidor puede provisionar unilateralmente capacidades de computación, como la hora del servidor y almacenamiento en red, automáticamente según su necesidad sin tener que haber una interacción humana con cada proveedor de servicio.

#### 2.1.2.2 Amplio acceso a la red

Las capacidades de los recursos de computación están disponibles a través de la red y accesibles mediante mecanismos estándar que promueven ser utilizados por plataformas de clientes heterogéneas, grandes o pequeñas.

#### 2.1.2.3 Agrupación de recursos

Los recursos de computación del proveedor están agrupados para servir a múltiples consumidores usando un modelo *multi-tenant*, con diferentes recursos físicos y virtuales asignados dinámicamente, y reasignados según la demanda del consumidor. Hay una

sensación de independencia de localización en la que generalmente, el cliente no tiene control o conocimiento sobre la exacta localización de los recursos proporcionados, pero puede tener la posibilidad de especificar la localización a un mayor nivel de abstracción.

#### 2.1.2.4 Rápida elasticidad

Las capacidades de los recursos de computación pueden ser provisionados y publicados elásticamente, en algunos casos automáticamente, para escalar rápidamente hacia fuera (*outward*) o hacia dentro (*inward*) dependiendo de la demanda. Para el consumidor, las capacidades disponibles para provisionar a menudo parecen ser ilimitadas y pueden ser asignadas en cualquier cantidad en cualquier momento.

#### 2.1.2.5 Servicio medido

Los sistemas *Cloud* controlan y optimizan automáticamente el uso de recursos aprovechando una capacidad de medición en algún nivel de abstracción apropiado para el tipo de servicio. El uso de recursos puede ser monitorizado, controlado e informado, proporcionando transparencia para el proveedor y consumidor del servicio utilizado.

### 2.1.3 Modelos de servicio

En este apartado se va a ver una descripción breve de los modelos de servicios explicados por el NIST [4], añadiendo dos más que han aparecido en los últimos años que son objeto de interes para el estudio y la aplicación que se hará más adelante. Se entrará en más detalle de cada uno de los modelos de servicio en el apartado 2.1.3 dónde se comparan los distintos modelos de servicio.

#### 2.1.3.1 Infraestructura como servicio (*IaaS*)

[4] El consumidor es capaz de provisionar procesos, almacenamiento, red, y otros recursos de computación fundamentales donde el consumidor es capaz de desplegar y ejecutar software arbitrario, que puede incluir sistemas operativos y aplicaciones. El consumidor no gestiona o controla la infraestructura *Cloud* subyacente, pero tiene control sobre los sistemas operativos, almacenamiento, y despliegue de aplicaciones; y posiblemente un control limitado de componentes de red seleccionados.

#### 2.1.3.2 Plataforma como servicio (*PaaS*)

[4] El consumidor es capaz de desplegar en la cloud infraestructura creada por él o aplicaciones adquiridas por el consumidor creadas utilizando lenguajes de programación, librerias, servicios, y herramientra que sean soportadas por el proveedor. El consumidor no gestiona o controla la infraestructura subyacente incluyendo redes, servidores, sistemas operativos, o almacenamiento, pero tiene control sobre el despliegue de aplicaciones y posiblemente las configuraciones para el entorno de alojamiento de aplicaciones.

#### 2.1.3.3 Software como servicio (*SaaS*)

[4] El consumidor es capaz de usar las aplicaciones de proveedores ejecutadas en una infraestructura *Cloud*. Las aplicaciones son accesibles desde varios dipositivos cliente a través de una interfaz de cliente ligera, como un navegador web, o una interfaz del programa. El consumidor no gestiona o controla la infraestructura *Cloud* subyacente incluyendo redes, servidores, sistemas operativos, almacenamiento, o incluso las capacidades de las aplicaciones individuales, con la posible excepción de la configuración de aplicación específica de usuario.

#### 2.1.3.4 Contenedores como servicio (*CaaS*)

El consumidor es capaz de provisionar procesos, almacenamiento, red, y otros recursos de computación virtuales donde el consumidor es capaz de desplegar y ejecutar software arbitrario, que puede incluir sistemas operativos y aplicaciones en contenedores con un nivel de virtualización de sistema operativo. El consumidor no gestiona o controla la infraestructura *Cloud* subyacente, pero tiene control sobre los sistemas operativos del contenedor, no del de la máquina virtual, almacenamiento, y despliegue de aplicaciones; y control limitado de componentes de red virtuales dentro del sistema operativo.

A diferencia de las máquinas virtuales de *IaaS* los contenedores de *CaaS* no necesitan un sistema operativo completo para ser ejecutadas. En definitiva, abstraen las aplicaciones del sistema operativo subyacente. Este concepto está muy ligado a la arquitectura de microservicios, que se verá con mayor detalle más adelante en el apartado 2.2.2.

#### 2.1.3.5 Funciones como servicio (*FaaS*)

El consumidor es capaz de desplegar en la cloud infraestructura creada por el consumidor o aplicaciones adquiridas por el consumidor creadas utilizando lenguajes de programación, librerias, servicios, y herramientra que sean soportadas por el proveedor. El consumidor no gestiona o controla la infraestructura subyacente incluyendo redes, servidores, sistemas operativos, o almacenamiento, pero tiene control sobre el despliegue de aplicaciones y posiblemente las configuraciones para el entorno de alojamiento de aplicaciones.

A diferencia de PaaS solo se despliega código o funciones que se ejecutan bajo eventos y está ligado al concepto de arquitectura *serverless*.

### 2.1.4 Economía de *Cloud Computing*

[8] Existen tres casos de uso que favorecen la utilización de *Cloud Computing* sobre una más convencional. Estos son:

- La demanda de un servicio varia en función del tiempo

Permite poder pagar por lo que realmente se consume en función de los picos de demanda, consiguiendo ajustar los recursos a las necesidades evitando el sobredimensionamiento. En definitiva, ajustar los recursos a la demanda.

- La demanda es desconocida

Parecido al caso anterior, pero sin conocer realmente la demanda que pueda haber sobre un servicio, teniendo picos o bajos donde se pueden ajustar los recursos de computación en función de la demanda real desconocida.

- Procesos en lotes:

Este sería el caso de convertir CapEx a OpEx, o lo que es lo mismo, el concepto de *pay as you go.* En este caso, se necesitará lanzar una serie de procesos en lotes que no necesitan de recursos de computación, mientras no se estén procesando.

En base a estos casos de uso, se va a poder ver como los distintos modelos de servicio pueden ser aplicados para ajustar los recursos de computación a la demanda. Esto se verá más adelante cuando se comparen cada uno de los modelos de servicio.

Por lo tanto, el concepto de *pay as you go* puede parecer más caro que comprar un servidor durante el mismo periodo, aún así el coste se ve superado por los beneficios extremamente importantes de la economía de *Cloud Computing* de elasticidad y transferencia de riesgos, principalmente los riesgos de sobredimensionamiento o infradimensionamiento.

En base a la elasticidad, el punto clave es la habilidad de *Cloud Computing* de añadir o eliminar recursos según las necesidades reales y con un tiempo de respuesta de minutos en vez de semanas. Esto permite ajustar de una manera más precisa los recursos a la carga de trabajo. Por lo tanto, esto permite poder reducir los riesgos de sobre/infradimensionamiento, adaptándose a la carga de trabajo reduciendo los costes del recurso de computación.

Aún así, hay que añadir que esto conlleva un riesgo añadido debido a que, si no se establece un control y ajuste del escalado, ya sea hacia fuera (*scale out*) o hacía dentro (*scale in*), puede implicar un coste añadido significante. Esto implica la necesidad de establecer soluciones "*Cloud Native*" que permitan aprovechar las ventajas que te da la *Cloud.* Si esto no se hace al final puede acabar siendo contraproducente y provocar sobrecostes inesperados.

En las siguientes figuras se pueden ver tres casos que pueden ocurrir si no se establecen políticas de escalado correctas que se anticipen a los cambios de demanda, y por lo tanto carga de trabajo, que puedan ir apareciendo a lo largo del tiempo.

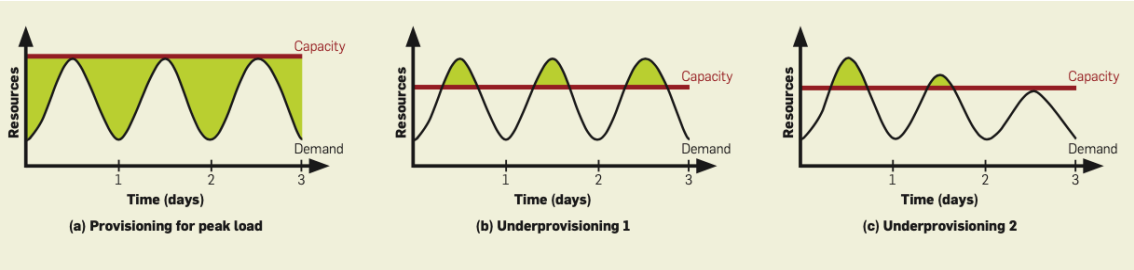



**Figura 1: [8] Perdida de recursos de computación por falta de elasticidad**

En la *Figura 1* se puede comprobar las consecuencias de no tener elasticidad, donde es necesario realizar un aprovisionamiento previo en función de la carga de trabajo y demanda estimada. En el primero (a) se ve como, que, a pesar de que se consiga predecir de manera adecuada los picos de utilización, se desperdicia mucha capacidad de computación que genera un sobrecoste. Así mismo, en el (b) se ve que si la dimensión estimada está por debajo de los picos esto va a provocar una saturación de los sistemas que implicará una degradación o caídas de los servicios.

Por lo tanto, sin un sistema elástico se corre el riesgo de tener perdidas al no aprovechar los recursos libres y que haya una saturación si no se han estimado los picos de manera correcta.

[1] El valor, generalmente, es algo que se obtiene cuando se compensan los beneficios de un servicio por su coste. Si a la suma total de beneficios se le resta los costes de *Cloud Computing,* se consigue una suma tangible que connota valor. El valor, por lo tanto, abarca un análisis de coste-beneficio, ya que, esencialmente se calcula como beneficios menos costes.

En *Cloud Computing,* un modelo de valor, es un patrón estándar que define el valor para el usuario o consumidor. A continuación, vamos a repasar varios modelos de valor que pueden verse en [1] y es interesante revisarlos.

#### 2.1.4.1 Gastos de operación (OpEx)

Es necesario invertir en capital de antemano para poder crear y mantener un servicio siendo necesario generar capacidad de computación extra para asegurar los acuerdos de nivel de servicio o *SLAs (Service Level Agreements)*. Todo este CapEx invertido representa un coste de oportunidad, ya que, ese capital podría emplearse en otros servicios que generen oportunidades de negocio extras. Así mismo, si no se ha estimado correctamente la capacidad no se podrá responder a la creciente demanda o se verán degradados los *SLAs*.

Ambos casos pueden provocar grandes perdidas ya sea por no satisfacer la demanda o por no tener recursos infrautilizados, que generan un coste perdido. Por lo tanto, con el modelo de valor OpEx que se puede ver en la *Figura 2* utilizando elasticidad se pueden evitar los costes de oportunidad y perdidas. Se asume que la demanda del negocio crecerá y que tendrá mayor automatización de escalabilidad *Cloud Computing*.

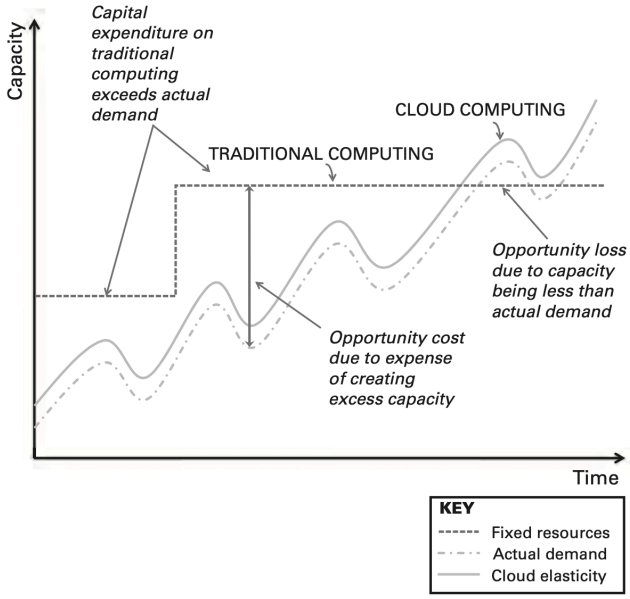



**Figura 2: [1] Modelo de valor de gastos de operación**

### 2.1.4.2 Flexibilidad de demanda de usuario

Los productos y servicios de un negocio proporcionan a los clientes unos ciclos de vida de desarrollo, testeo, lanzamiento, marketing y uso de negocio normal. En cada una de estas fases, son empleados en diferentes demandas para la computación por lo que habrá periodos que consisten en picos durante todo el ciclo de vida. Para asegurar que se cumplen estas demandas, será necesario invertir y desinvertir en sobredimensionar capacidad de computación. Por lo tanto, habrá periodos en el que la demanda no se pueda satisfacer. El modelo de flexibilidad de demanda que se ve en la *Figura 3* usa elasticidad para asegurar que las distintas demandas de computación se puedan satisfacer en los distintos ciclos de vida del producto.

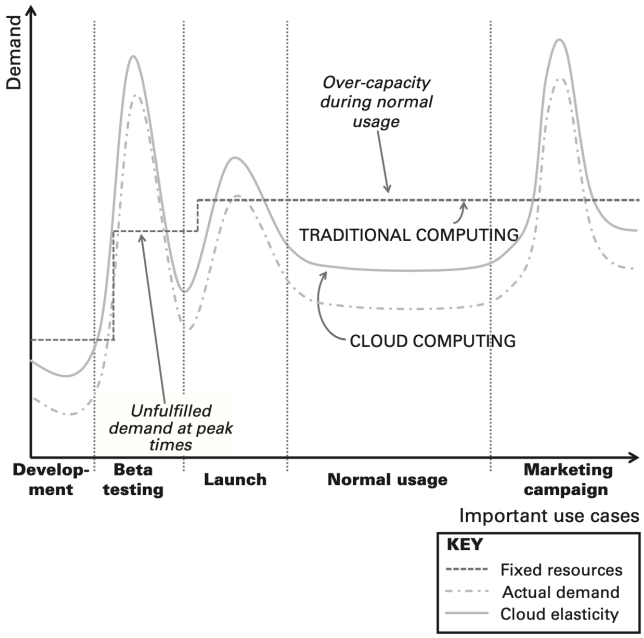



**Figura 3: [1] Modelo de valor de flexibilidad de demanda de usuario**

### 2.1.4.3 Flexibilidad de precio

En general, el precio de computación tiende a reducirse debido a que las nuevas tecnologías emergen rápida y regularmente. Otro factor que contribuye a la tendencia de bajada de precios, es la economía de escala, que resulta en la adopción de nuevas tecnologías al quedarse obsoletas las antiguas. *Cloud Computing* permite introducir una bajada en escalón en la bajada de la curva de precios, utilizando distintos modelos de precios en lo que las necesidades y requerimientos de volumen varían. Teniendo la posibilidad de cambiar el modelo de precios permite obtener mayor valor del dinero. Por lo tanto, el modelo de valor de flexibilidad de precio, como se puede ver en la Figura 4, usa diferentes modelos de precio para reducir los costes de computación en función de la demanda, volumen y cambios circunstanciales.

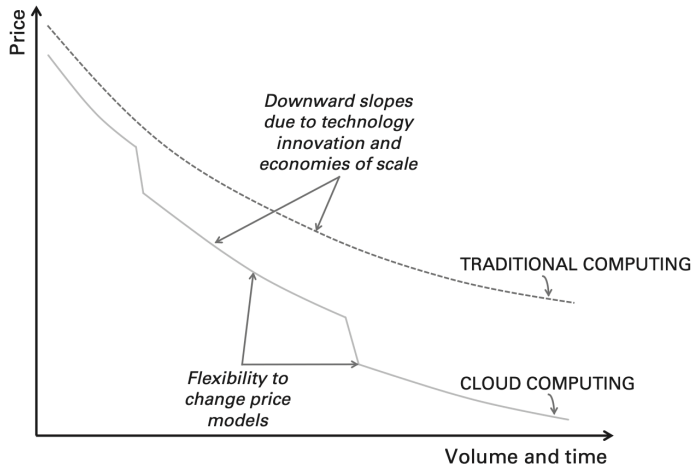



**Figura 4: [1] Modelo de valor de flexibilidad de precio**

### 2.1.4.4 Agilidad de *Time to Market*

El modelo de valor de *time to market* de *Cloud Computing*, como se puede ver en la Figura 5, reduce el tiempo de salida a mercado habilitado la transición de un entorno de computación a otro con mayor velocidad. Esto se debe, a que no se tiene que estar perdiendo tiempo en generar infraestructura de computación y entornos de antemano, se pueden generar según la necesidad.

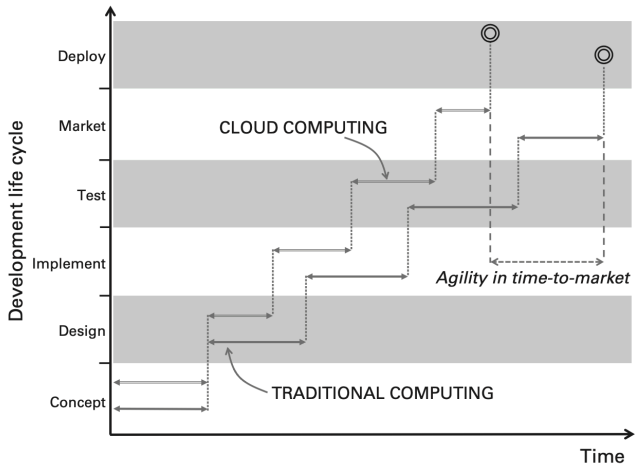



**Figura 5: [1] Modelo de valor *time to market***

### 2.1.4.5 Flexibilidad de localización

El modelo de valor de flexibilidad de localización, como se puede ver en la *Figura 6*, permite acceder a los entornos de computación y trabajar en cualquier localización. Está flexibilidad incrementa la productividad y permite tener un acceso global.

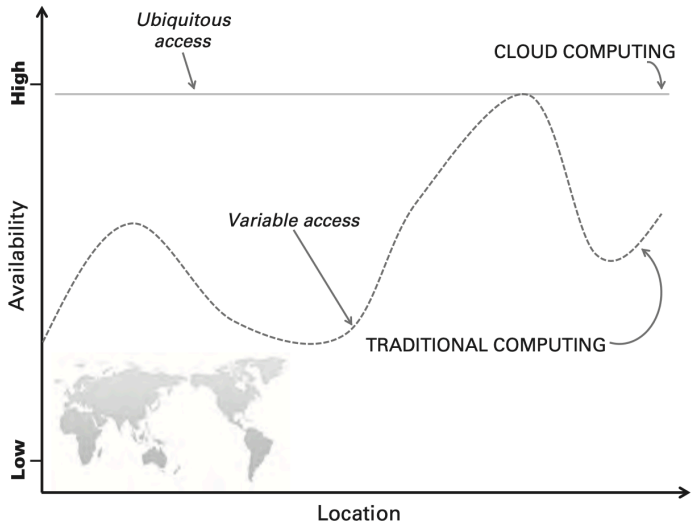



**Figura 6: [1] Modelo de valor de flexibilidad de localización**

### 2.1.4.6 Optimización de activos

El modelo de valor de optimización de activos de *Cloud Computing*, como se puede ver en la *Figura 7,* es similar al modelo de valor de flexibilidad de demanda, con otra perspectiva. En este caso, se consideran las inversiones extra que serán necesarias realizar para poder tener capacidad en exceso para prevenir perdidas de negocio.

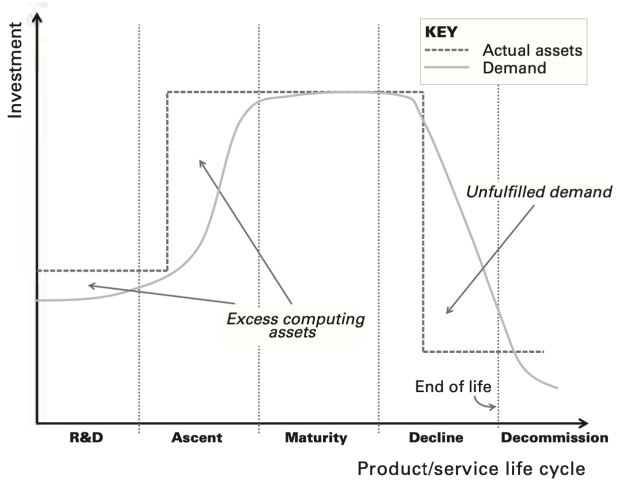



**Figura 7: [1] Modelo de valor de optimización de activos**

### 2.1.4.7 Margen de beneficios

Cuando se produce cualquier bien o servicio, los costes de valor añadidos caen un 25% cada vez que el volumen acumulado, o la experiencia acumulada, se dúplica. Esto implica una bajada de la pendiente de curva de precios que incrementa el beneficio en base a la experiencia. La curva de precios es conocida como la curva de experiencia. *Cloud Computing* es un cambio evolucionario de la computación tradicional, ya que se basa en la experiencia acumulada de la computación. Como tal, representa un desplazamiento hacía la derecha de la curva de experiencia de computación tradicional, como se puede ver en la *Figura 8.*

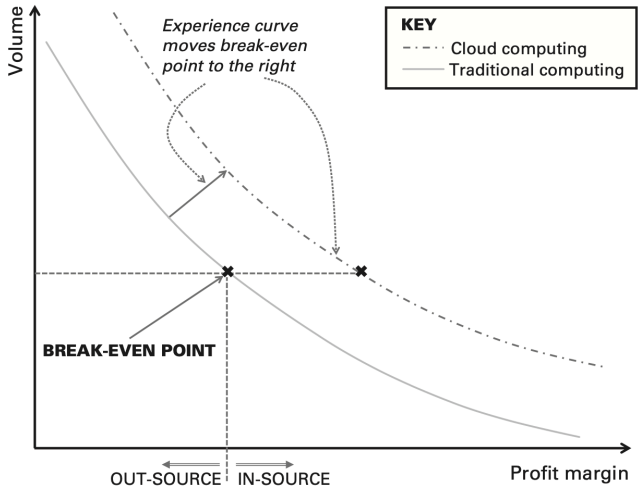



**Figura 8: [1] Modelo de valor de margen de beneficios**

### 2.1.4.8 Métricas financieras

Por último, se ha de trasladar el valor que proporciona *Cloud Computing* a una serie de medidas financieras significativas como:

- Método de amortización, mide el tiempo necesario para amortizar la inversión de un producto o servicio, cuanto más pequeño sea mejor. Para cubrir esto, se suelen utilizar cálculos de valor presente neto o *NPV*

- Retorno de inversión o *ROI*, a diferencia del método de amortización usa el porcentaje de la cantidad de inversiones que serán recuperados.

  ROI = (Ganancia de inversión – Coste de inversión) / (Coste de inversión)

- Valor presente neto o *NPV,* está basado en conjuntos de *cash flows* siguiendo la siguiente ecuación, con r siendo el ratio de descuento, N el número de meses y A el coste mensual.

$$NPV = \frac{A}{\sum_{k=0}^{N}(1+r)^{k}}$$

- *Time to market (TTM)*, estima el tiempo en el que se empezaran a obtener ingresos por la puesta en el mercado del producto o servicio.

### 2.1.5 Ventajas de *Cloud Computing*

[8] Se pueden resumir todas las ventajas de *Cloud Computing* en:

- Eficiencia y reducción de costes, se reduce la inversión en comprar y mantener equipos, reduciendo *CapEx*.

- Seguridad de datos, los sistemas *Cloud* permiten tener elementos de seguridad avanzando que permite almacenar y gestionar los datos de manera segura.

- Escalabilidad, permite dimensionar la infraestructura en función de las necesidades reales, permitiendo minimizar riesgos asociados por sobre o infra dimensionamiento.

- Amplio acceso y disponibilidad, permite acceder a los recursos de manera sencilla desde múltiples vías de acceso y asegura poder tener alta disponibilidad por replicación.

- Actualizaciones automáticas, no necesita tener un mantenimiento constante y se realizan actualizaciones automáticas de los servicios *Cloud.*

- Recuperación ante desastres, al ser sistemas distribuidos permite tener preparado para replicar y mantener los datos y sistemas en caso de desastre.

- Control, te permite tener una visibilidad y control completo de todos los datos y sistemas, incluido tenerlas versionadas.

- Estrategia competitiva, permite poder adaptar y realizar cambios de una manera más ágil según las necesidades del negocio. Pudiendo adaptarse a los cambios tecnológicos y necesidades futuras.

### 2.1.6 Comparación de modelos de servicio

La manera más sencilla de poder hacer una comparación eficaz de cada uno de los modelos de servicios es visualizar la *Figura 9* [8], donde se ven los componentes de cada uno de los modelos.

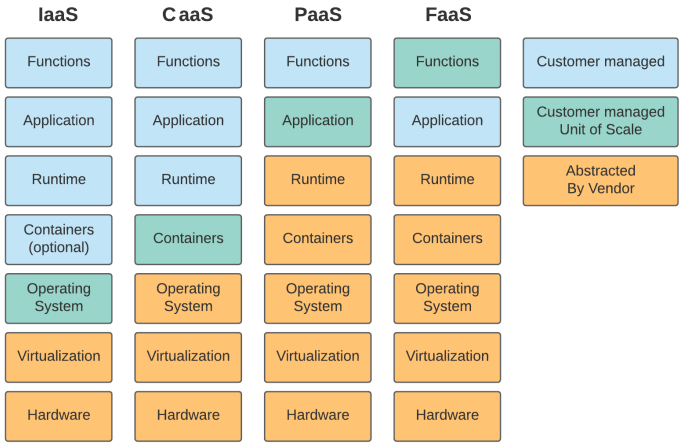



**Figura 9: Comparación de modelos de Servicio**

Para esta comparación hay que entender que el contexto es buscar el mejor modelo de servicio para una solución IoT, queda descartado el modelo de servicio *SaaS,* ya qué este tipo de modelo no te permite cambiar características del servicio al dar ya una aplicación y un servicio completo.

Se puede observar que de izquierda a derecha hay un cambio de menor a mayor nivel de abstracción de la infraestructura y sistemas operativos subyacentes. Para cada tipo de servicio se van añadiendo capas dónde se pueden aplicar cambios. *IaaS* es el más configurable, pero a la vez más tedioso, ya que implica tener una mayor dificultad de mantenimiento y una buena configuración para optimizarlo según el software que se ejecute en el mismo.

Ha diferencia de *IaaS*, *CaaS* permite olvidarse del sistema operativo subyacente y se centra en la virtualización de este. Habitualmente es necesario utilizar un orquestador de contenedores como Kubernetes o Docker Swarm, en el apartado *16* se verá con más detalle. Esto nos permite ejecutar contenedores desplegados como microservicos que permiten tener una arquitectura orientada a servicios o *SOA* (*Service Oriented Architecture)* con contenedores distribuidos a través de distintos nodos de computación. Esto implica una menor necesidad de configuración a nivel de sistema operativo, pero a la vez es necesario diseñar, gestionar y mantener los distintos componentes de cada contenedor.

El siguiente paso sería *PaaS* que para este caso solo es necesario desplegar la propia aplicación, olvidándose de la gestión de los contenedores. Pudiendo desplegar directamente la aplicación sin necesidad de tener que gestionar la infraestructura subyacente. El problema principal de este tipo de modelo de servicio es que viene limitado por los lenguajes de programación que hay disponible. Aún así, genera una ventaja al añadir una capa de abstracción más, reduciendo el mantenimiento y configuración necesaria. Con este modelo de servicios automáticamente escala, distribuye los procesos y mantiene una alta disponibilidad. Se sigue pudiendo elegir los tamaños de computación y los límites de computación a establecer.

Finalmente, llegamos a la máxima expresión de abstracción con FaaS donde aparece una diferencia principal, no se despliegan microservicios si no funciones que se ejecutan bajo eventos. Es decir, se basa en una arquitectura bajo eventos donde se abstrae completamente de toda la infraestructura subyacente. Se pueden hacer configuraciones básicas, pero olvidándose de la gestión de los escalados, disponibilidad, despliegues y configuraciones de recursos. Aquí es donde llega el concepto de arquitectura *serverless* al tener tal capa de abstracción que llega el punto que no necesitas saber las características de los recursos subyacentes.

Esto permite que sea un servicio con auto-escalado completo y solo se pague por el consumo real, ya que no se reservan recursos de computación. También se debe a que las funciones solo se ejecutan cuando ha habido un evento y, en caso contrario, se mantienen “apagadas” sin consumir recursos. Otra característica para tener en cuenta es que deben ser ejecuciones en tiempos mínimos, entre 5 y 10 minutos, ya que las funciones deberán consumir recursos el mínimo tiempo posible. En caso contrario es necesario dividir esa función en funciones más pequeñas para distribuir los procesos.

[9] En base a las características de los distintos modelos de servicio, se puede ver que en *FaaS* la lógica es, desarrollada dentro de un flujo de trabajo clásico que automáticamente es desplegada en procesos sin estado que son capaces de responder a distintos eventos que se produzcan en la infraestructura, con tiempo de vida limitado y completamente gestionada por el proveedor. En base a esto se puede observar como *FaaS* encaja perfectamente con una solución IoT debido a su característica *serverless,* que se explorará con mayor detalle en los siguientes apartados. Por lo tanto, a partir de aquí en adelante se utilizará *FaaS* como el modelo de servicio de referencia.

## 2.2 *Arquitectura Serverless*

Una de las mejores definiciones de una arquitectura serverless se puede encontrar en la siguiente cita [9]:

"En este modelo, la computación pasa finalmente a ser como la luz o el agua, una 'utility' que consumiremos en función de nuestras necesidades"

Creo que esta definición deja claro las ventajas y características de las arquitecturas *Serverless* y, por lo tanto, el modelo de servicio *FaaS.* El término *serverless* [9] es el modelo de computación según el cual el proveedor de la capa de computación nos permite ejecutar durante un periodo de tiempo determinado, cuando un evento lo dispara, porciones de código denominadas "funciones" sin necesidad de hacernos cargo de la infraestructura subyacente que se provisiona para dar el servicio.

Se puede ver que en este modelo el proveedor se encarga de ofrecer los recursos de forma transparente, de escalarlos automáticamente si crece la demanda y de liberarlos cuando no son utilizados. Por lo tanto, se definen una serie de restricciones de procesamiento y un modelo de pago por el consumo de recursos por ejecución. Típicamente una función no debería ejecutarse durante más de 5 o 10 minutos por ejecución.

### 2.2.1 Características principales

Las características principales de una arquitectura *serverless* son [9]:

- Sin administración de infraestructura
- Auto-escalado
- Pago por uso
- Reducción del time-to-market
- Tiempos cortos de ejecución

### 2.2.2 Tipos de arquitectura

Según la arquitectura *backend* se pueden encontrar actualmente tres tipos de diseño:

- Monolíticos, está desplegado en un único sitio por lo que se vuelven pesados y difíciles de escalar. Normalmente se despliegan sobre *IaaS* u *on-premise*. Este tipo de arquitectura, generalmente se podría considerar que no es *Cloud native* al no explotar todas las ventajas de *Cloud Computing.*

- Microservicios, el *backend* se divide en diversos microservicios, habitualmente entre 10 y 80, que pueden ser desplegados de manera distribuida. Cada uno de los microservicios debería realizar una función de negocio independiente.

- *Serverless* o funciones, a diferencia de las dos anteriores, no tienen que estar ejecutándose todo el tiempo solamente cuando se produzca un evento que las ejecute. Esto implica que se convierte en una arquitectura basada en eventos que encaja perfectamente con soluciones IoT, donde solo se ejecutaran cuando son necesarias.

En la Figura 10 se puede observar mejor las diferentes arquitecturas.

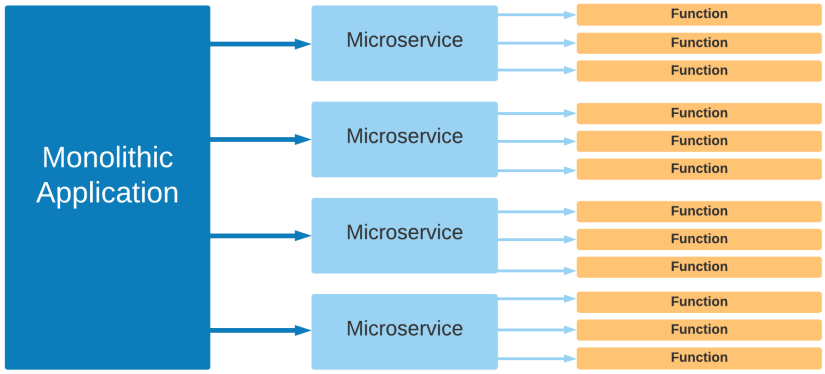



**Figura 10: Arquitecturas de aplicación backend**

### 2.2.3 Origen

[10] Los sistemas actuales de *software* web se componen de dos componentes principales, por un lado, los servidores *backend* que realizan los distintos procesos de computación y, por otro lado, el *frontend* que proporciona una interfaz para que los usuarios puedan operar mediante un navegador, móvil u otro dispositivo.

Los sistemas se vuelven más complejos, una vez se tiene en cuenta elementos como los balanceadores de carga, transacciones, clusters, cacheo, mensajería, y redundancia de datos. La mayoría de software requiere servidores jecutados en centros de datos o en la nube que necesitan ser gestionados, mantenidos, parcheados, y realizar *back up.*

Todos estos procesos se converiten tareas que acaba consumiendo mucho tiempo y requieren equipos dedicados para operarlos constantemente. Además, hay que tener en cuenta que los entornos complejos son díficiles de configurar y operarar de manera eficiente. La infraestructura y el hardware son componentes necesarios de cuaqluier sistema IT, pero al mismo tiempo son una distracción de lo que debería ser el foco central, resolver problemas de negocio.

A lo largo de los útlimos años, tecnologías como *PaaS* y *CaaS* han aparecido como solcuiones potenciales para resolver los quebraderos de cabeza de entornos de infarestructuras incosistentes, conflictos, y sobrecostes de gestión de servidores. Como ya se ha visto, *PaaS* es una forma de computación *Cloud* que proporciona una plataforma para que los usuarios puedan ejecutar su *sofware* ocultando ciertos aspectos de la infraestructura subyacente. Para realizar un uso efectivo del *PaaS*, los desarrolladores han de adaptar el software para que se beneficie de las características y capacidades de la plataforma.

Por lo tanto, eliminar aplicaciones *legacy* diseñadas para ejecutrase en servidores *standalone* a un servicio *PaaS* implica un esfuerzo a nivel de desarrollo por la naturaleza efimera de las implementaciones *PaaS.* Aún así, los beneficios que consiguen debido a la reducción de los mantenimiento y los requisitos de soporte de la plataforma son considerables a medio y largo plazo.

La contenerización o *CaaS* es una forma de aislar una aplicación de su propio entorno. Es una alternativa ligera de la virtualización completa donde se hace una virtualización a nivel de sistema operativo. Sin embargo, a pesar de que estos contenedores están asilados y son ligeros necesitan ser desplegados en un servidor independientemente, sean entornos *Cloud* o no. Son una solución excelente cuando hay problemas de dependencias, pero pueden aparecer una serie de complejidades y retos en cuanto a su gestión. No es tan sencillo como simplemente ejecutar código en la *Cloud.*

Finalmente, aparece la solución *serverless* o *FaaS* que son servicios de computación en la *Cloud* que permiten ejecutar códigos de manera paralizada bajo eventos. Este tipo de servicios coje el código y lo ejecuta sin necesidad de tener que provisionar servidores, instalar software, desplegar contenedores, o preocuparse de detalles a bajo nivel. Estos servicios por si mismos, se encargan de provisionar y gestionar los componentes *IaaS* que ejecutan el código y proporciona infraestructura de computación con alta disponibilidad, incluyendo capacidades de aprovisionamiento y escalado automático del cual los desarrolladores se abstraen completamente.

El concepto de arquitectura *serverless* se refiere a estos nuevos modelos de arquitectura de *software* que no dependen de acceso directo a servidores para funcionar. Escogiendo un servicio *FaaS* y utilizando sus diversas funcionalidades se pueden desplegar rápidamente arquitecturas desacopladas, escalables y eficientes. Alejarse de los problemas de servidores e infraestructuras, y al mismo tiempo permitir que los desarrollos se centren en la lógica del código reduciendo la gestión que implica mayor agilidad de cambios y despliegues, y por lo tanto, reducción del *time to market* es el objetivo detrás de la arquitectura *serverless.*

### 2.2.4 Arquitectura Orientada a Servicios o *SoA*

[10] La arquitectura orientada a servicios o *SOA (Service Oriented Architecture)* conceptualiza la idea de que un sistema puede estar compuesto por multiples servicios independientes. Este tipo de arquitectura no dicta el uso de una tecnología en particular, al contrario, fomenta un enfoque arquitectural en la que se creen servicios autónomos que se comunican mediante el intercambio de mensajes. La reusabilidad y autonomia, componibilidad, granularidad, y descubrimiento de los servicios son principios importantes asociados con *SOA.*

Los microservicios y arquitecturas *serverless* heredan el concepto de arquitectura orientada a servicios. Estos mantienen en gran medida los principios e ideas de este tipo de arquitecturas mientras intentan abordar la complejidad de SOA más *legacy.*

Recientemente hay una tendencia a implementar sistemas con microservicios, en la que se tiende a pensar que son pequeños servicios completamente independientes que se construyen alrededor de un propósito o una capacidad de negocio.

Idealmente, los microservicios deberían ser sencillos de remplazar, con cada uno de los servicios escritos con el *framework* y lenguaje adecuado. El mero hecho de que los microservicios pueden ser escritos en diferentes lenguajes de dominio específicos o *DSL (Domain-Specific Languages)* es un factor diferenciador. Se pueden obtener diversos beneficios mediante el uso del lenguaje adecuado o un set de librerias especializadas para determinada tarea. Aún así, tener una mezcla de lenguajes y frameworks puede ser difícil de mantener y, a la larga, puede generar problemas sin una disciplina estricta.

Cada microservicio puede mantener un estado y almacenar datos, si además están correctamente desacoplados se puede trabajar y desplegar los microservicios independientemente uno de otro. A pesar de ello, la consistencia eventual, gestión de transacciones, y recuperación de errores complejos puede acabar provocando que sea más difícil.

Se puede decir que las arquitecturas *serverless* conllevan multiples principios de los microservicios. Después de todo, dependiendo como se diseñan los sistemas, cada función ejecutada puede considerarse como su propio servicio independiente. Aún así, no es necesario llevar un concepto de microservicos en este tipo de servicios si no es necesario.

En definitiva, las arquitecturas *serverless* permiten la libertad de aplicar la cantidad de principios de los microservicios que se consideren necesarios sin forzar una única forma arquitectura de diseño. Por lo tanto, dando mayor flexibilidad y libertad de diseño según las necesidades que surjan. Por ejemplo, se puede intuir que una arquitectura de una solución serverless *IoT* variara considerablemente con respecto al de una aplicación web más clásica.

### 2.2.5 Contenedores y Orquestadores

Antes de nada, para entender las bases de la arquitectura *serverless* y en concreto *FaaS* primero hay que entender los conceptos de contenedores y los orquestadores de contendores. Esto se debe a que es interesante conocer conceptualmentelas tecnologías que han permitido la aparición de la arquitectura *serverless.*

#### 2.2.5.1 Contenedores

Este apartado se va a centrar en una de las tecnologías más populares de software de contendores de los últimos años, concretamente Docker, aunque existen otras soluciones como Cri-o que se consideran igualmente validas ya que conceptualmente no hay mucha diferencia.

[13] Docker ofrece una virtualizacuón a nivel de sistema operativo o *OS* (Operative System), conocido como contenerización. Debido a este tipo de virtualización es posible ejecutar más de un sistema operativo dentro de otro.

La diferencia principal entre un contenedor y una máquina virtual es que, el contenedor no necesita un *OS* completo para poder ejecutarse. Es decir, cuando se crea una máquina virtual, se recrea un sistema operativo completo, sin embargo, con un contenedor solo se virtualiza parte del sistema operativo. Por lo tanto, se consigue reducir el tamaño de la imagen.

En las virtualizacion de *OS*, al hacer una virtualización a nivel de sistema operativo el host aisla cada contenedor uno del otro. Solo se pueden ejecutar contenedores con el mismo host, es decir, no se es posible ejecutar un Windows en un host de Linux, ya que la virtualización es ejecutada por el sistema. Esto se debe a que Windows y Linux tienen dos tipos diferentes de kernel de *OS* y estructura de sistema de archivos.

Algo a tener en cuenta sobre los contenedores es que tienen menos seguridad comparado con una virtualización de hipervisor. Esto se debe a que no utiliza librerías de que mejoran la seguridad del *OS* como por ejemplo, SELinux y cgroups. A pesar de ello, aunque uno de los contenedores se vea compremetido es probable que no genere un problema en el sistema host, debido a que los contenedores son entornos aislados.

Docker añade una capa de despliegue de aplicación sobre el contenedor virtualizado. Por lo tanto, se puede consiguir un entorno de ejecución completamente similar al de producción, que permite acelear los procesos de desarrollo, reducir el time to market y evitar sopresas debido a diferenciación entre entornos.

A nivel arquitectural, con Docker, es sencillo implementar una arquitectura de microserficios. Esto se debe a que cualquier contenedor puede ser una pieza única de una aplicaciín, y se puede gestionar esta pieza de manera independiente.

#### 2.2.5.2 Orquestador de Contenedores

Este apartado se va a centrar en una de las tecnologías más populares de orquestadores de contendores de los últimos años, concretamente Kubernetes, aunque existen otras soluciones como Docker Swarm que se consideran igualmente validas, ya que conceptualmente no hay mucha diferencia.

Con la aparición de contenedores surge la necesidad de poder orquestar la gestión, automatización de despliegues y escalado de las aplicaciones contenerizadas. Esto se debe a que los contenedores por si solos son difíciles de mantener y una orquestación adecuada permite mejorar estos procesos.

[13] Entre los diversos orquestadores de contenedores, contretamente Kubernetes es comunmente utilizado para la gestión y el escalado de aplicaciones contenerizadas en un cluster, como Docker.

Kubernetes define bloques básicos para crear y gestionar un recurso, donde todos los componentes están diseñados para estar desacoplados. El componente principal de Kubernetes es conocido como pod, que es una colección de contenedores Docker, todos

con la mismas direcciónes IP. Cada pod está compuesto por uno o más contenedores, ejecutados en un mismo host y compartiendo recursos. Cada pod compoarte una unica dirección de IP con el cluster que puede ser gestionada manulamente, por la API o el controlador.

Otro recurso definido en Kubernetes son las etiquietas o *labels*, que se utilizan para crear un par de clave-valor para identificar el recurso. Las etiquetas pueden ser asignadas a más de un objecto con el mismo nombre para crear lo que se conoce como un selector de etiquetas o *label selector,* que permite identificar un conjunto de objetos

El controlador es usado para gestionar el estado del clúster*,* para cambiar el estado del clúster el controlador gestiona el *pod*. Hay tres tipos de controladores:

- Replicación, se utiliza para replicar el servicio a través del clúster de Kubernetes. También permite rotar nuevos *pods* cuando uno falla permitiendo mantener el mínimo número de nodos corriendo y asegurar alta disponibilidad en el clúster.

- *DaemonSet*, es el responsable de asegurar que todos los conodos ejecutan una copia de un *pod* cuando se añade un nodo al clúster. Los *pods* se añaden a este nuevo nodo mediante este servicio.

- *Jobs,* permite ejecutar un *pod* hasta que uno o todos se ejecutan completamente de manera satisfactoria.

Por último, están los servicios que son un conjunto de *pods* que trabajan conjuntamente. Es decir, se puede crear un *pod* con una pieza completa de la arquitectura de microservicios y se define un servicio. Los servicios se definen e identifican con una etiqueta que permite identificar el servicio. Por lo que es posible buscar el servicio con un descubrimiento de servicios y usar la dirección IP y el nombre DNS para identificar el servicio que se quiere gestionar.

Como ya se ha mencionado, Kubernetes es un orquestador de contenedores para la automatización de despliegues, gestión y escalado de aplicaciones contenerizadas que están definidos dentro de los *pods*. Esto permite un alto nivel de abstracción, y agrupación de aplicaciones contenerizadas.

Mediante Kubernetes se puede crear un clúster para gestionar el sistema, que en definitiva es una colección de nodos. Un clúster de Kubernetes tiene un nodo máster y cero o más nodos de trabajo o *workers.* Un nodo es una máquina física o virtual en la que es posible ejecutar y programar un *pod.*

Uno de los componentes clave de *Kubernetes* es el etcd. Es un almacén de claves ligero, distribuido y persistente desarrollado por CoreOS. Este componente se utiliza para almacenar el estado de un clúster en cualquier momento, se utiliza para definir un clúster de alta disponibilidad.

Debido a todas las características de un orquestador de contenedores, en este caso Kubernetes, es posible crecer junto con la necesidad de negocio debido a la escalabilidad que proporciona, ya que, es posible añadir o eliminar nodos del clúster, y Kubernetes se encarga de gestionar todos los *pods*.

### 2.2.6 Principios de la arquitectura *serverless*

[10] Se definen cinco principios de la arquitectura *serverless* que describen como un sistema *serverless* ideal se debería construir. Estos principios sirven de guía para tomar las decisiones a la hora de construir aplicaciones *serverless*.

#### 2.2.6.1 Usar un servicio de computación para ejecutar el código bajo demanda

Como ya se ha visto anteriormente, la arquietectura serverless es una extensión de las ideas que surgen en la arquitectura orientada a servicios o *SOA*. En la arquitectura serverless todo el código es desarrollado y ejecutado en funciones aisladas, independientes y granulares que se ejecutan en un servicio de computación sin estado.

Se pueden desarrollar funciones para realizar las tareas más comunes, como la lectura y ejecución de orígenes de datos, llamar a otras funciones y realizar cálculos. En casos más complejos, se pueden orquestar invocaciones de múltiples funciones.

#### 2.2.6.2 Desarrollar funciones sin estado de proposito único

Es importante intentar diseñar funciones con el principio de única responsabilidad o *SRP (Single Responsability Principle)*. Esto implica que una función que solo tiene una funcionalidad sea más testeable y robusta permitiendo reducir los bugs y efectos inesperados.

Combinando funciones y servicios en una arquitectura desacoplada, se puede construir sistemas de *backend* complejos que son comprensibles y fáciles de gestionar. Una función granular con una interfaz bien definida es más probable que sea reutilizada dentro de una arquitecutra *serverless.*

El código debe desarrollarse de una manera sin estado o *stateless*, que no asuma recursos locales o procesos que puedan mantenerse en otras ejecuciones. *Statelessness* es potente debido a que permite que la plataforma escale rápidamente para poder manejar un número de eventos que cambian constantemente.

#### 2.2.6.3 Diseño basado en eventos *push*

Las arquitecturas serverless se pueden construir para servir cualquier propósito. Los sistemas se pueden construir de manera *serverless* desde cero, o aplicaciones monolíticas se pueden gradualmente reingeniería para obtener las ventajas de este tipo de arquitectura. Los diseños *serverless* más flexibles y potentes están basados en eventos *push*, siendo este uno de los principios más importantes.

Este tipo de sistemas comúnmente reducirá costes, complejidad y potencialmente mejorar las distintas soluciones. También hay que tener en cuenta que este tipo de sistemas no tienen porque ser apropiados o, incluso, conseguibles en todos los casos de

uso. Ocasionalmente será necesario ejecutar las funciones que lance el origen de eventos o que se ejecute de manera programada.

#### 2.2.6.4 Crear *frontends* más robustos y potentes

Una de las cosas que hay que tener en cuenta es que el código que se ejecuta en *FaaS* ha de tener tiempos de ejecución bajos. Esto se debe a que funciones que tarden menos en ejecutarse son más baratas ya que el coste de este servicio viene determinado por el número de peticiones, la duración de la ejecución y la cantidad de memoria reservada.

En consecuencia, construir un *frontend* potente y robusto que pueda utilizar servicios de terceros directamente implicará una mejora de experiencia. Menos saltos entre los recursos online y reducción de latencias implicarán una mejora en el rendimiento de la aplicación. En otras palabras, no es necesario enrutar todo a través de los servicios de computación. El *frontend* debe ser capaz de poder comunciarse directamente con un proveedor, base de datos, o cualquier otra API que pueda resultar útil

Tokens de firma digital pueden permitir al los *frontends* hablar con diversos servicios, incluidos bases de datos, de una manera segura. Esto de manera opuesta a sistemas tradicionales donde todos los flujos de comunicación ocurren a través de los servidores *backend.*

Aún así, no todo es posible o se debería realizar en el *frontend*. Hay secretos que no se pueden confiar a los dispositivos cliente. En este caso, un servicio de computación se debe usar para coordinar las acciones, validar los datos, y cumplir la seguridad.

También, hay que considerar la consistencia, si el *frontend* es responsable de escribir en múltiples servicios y falla durante el proceso, puede dejar el sistema en un estado inconsistente. Para este caso, *FaaS* debería ser utilizado para poder manejar errores y reintentar si la operación falla.

#### 2.2.6.5 Adoptar servicios de terceros

Los servicios de terceros solo pueden proporcionar un valor adicional y reducir código customizado. Aún así, hay que tener en cuenta factores como el precio, capacidades, disponibilidad, documentación y soporte. Es importante no recrear funcionalidades ya existentes y reutilizar código, librerías o servicios de terceros permitiendo centrar todo el esfuerzo en la capa de negocio.

### 2.2.7 Ventajas y desventajas de este tipo de arquitectura

Hay una serie de ventajas de implementar sistemas completa o parcialmente serverless, incluido la reducción de costes y aceleración del *time to market.* De todas formas, hay que tener cuidado considerando la ruta hacía la arquitectura *serverless* en el contexto de creación de las soluciones.

#### 2.2.7.1 Ventajas

La arquitectura *serverless* permite centrarse en el diseño del *software* y el código abstrayéndose de la infraestructura. Es más sencillo conseguir escalabilidad y alta disponibilidad y el coste suele ser más justo ya que solo pagas por lo que realmente utilizas. Cabe destacar que con *serverless* aparece un potencial añdido que permite reducir algunas de las complejidades del sistema minimizando el número de capas y aumentado el código que se necesita.

- **No más servidores:**

Tareas como las de configuración, gestión, parcheado y mantenimiento son delegadas en el proveedor, que permite ahora tiempo y dinero. Siempre y cuando no se necesite una serie de requisitos específicos para gestionar o modificar recursos de computación. En definitiva, solo eres responsable de tu propio código, dejando las tareas de operación y administración en manos del proveedor.

- **Multitud de usos:**

La capacidad de no tener estados y escalabilidad de computación puede ser utilizada para solventar problemas que se benefician del procesamiento paralelo. Tareas que solían llevar semanas se pueden realizar en días u horas mientras se realice la correcta combinación de tecnologías. Una aproximación *serverless* puede funcionar excepcionalmente para *startups* que quieren innovar y avanzar rápidamente.

- **Bajo coste:**

Las arquitecturas tradicionales basadas en servidores requieren servidores que no necesariamente se ejecutaban con toda capacidad durante todo el tiempo. Escalado, incluso con sistemas automatizados, implica nuevos servidores, que normalmente se desperdician hasta que aparezca un nuevo pico de tráfico o nuevos datos. Los sistemas *serverless* son mucho más granulares en cuanto a escalado y efectividad de costes, especialmente cuando hay picos de carga desigual o inesperada.

- **Menos código**

La arquitectura *serverless* permite reducir algunas de las complejidades y el código comparado con sistemas tradicionales. Hay menos necesidad de tener un sistema *backend* multicapa, espcialmente si permites que el *frontend* realice gran parte del trabajo y se comunique directamente con servicios incluido bases de datos.

- **Escalable y flexible**

No es necesario que la arquitectura *serverless* remplace completamente el *backend*. Se puede utilizar las *FaaS* para realizar problemas específicos, especialmente si se benefician de la paralelización. Este tipo de arquitecturas escalan de manera más sencilla en comparación con sistemas tradicionales.

#### 2.2.7.2 Desventajas

- **No para todos**

Tienden a ser ejecutadas en *Cloud* públicas, por lo que aplicaciones o servicios críticos no necesariamente deberían estar basados en ellos. Cabe la posibilidad de que el rendimiento y fiabilidad que proporciona la *Cloud* pública no se adecue el tipo de aplicación. Siendo necesario utilizar *hardware* dedicado o ejecutar *Cloud* privada o híbrida con sus propios servicios de computación que cubran las necesidades de fiabilidad y mantemiento.

- **Niveles de servicio**

Las *Cloud* públicas tienen una serie de *SLA (Service Level Agreements)* para determinados servicios, pero no todos pueden afectar su adopción. Para la mayoría de los sistemas, la fiabilidad es suficiente pero algunos casos de negocio requerirán garantías adicionales.

- **Customización**

En cuanto a *FaaS*, la eficiencia de que tener al proveedor manteniendo la plataforma y escalando las funciones implica que no es posible customizar el sistema operativo o las instancias subyacentes. Se pueden modificar determinados parámetros como por ejemplo la memoria alocada o *timeouts* pero poco más.

- **Dependencia del proveedor**

*Vendor Lock in* o dependencia del proveedor es otro de los problemas que aparecen. Puede ocurrir que es acabe acoplando fuertemente a la plataforma y servicios del proveedor. Por lo tanto, hay que considerar la dependencia del proveedor y los riesgos de usar servicios de terceros incluyendo viabilidad de la compañía, soberanía de datos, privacidad, coste, soporte, documentación y el conjunto de funciones disponibles.

- **Descentralización**

La transición de soluciones monolíticas a soluciones *serverless* descentralizadas no reduce automáticamente la complejidad de los sistemas subyacentes. La naturaleza distribuida de la solución puede introducir retos propios debido a la necesidad de hacer llamadas remotas en vez de llamadas a procesos locales necesitando manejar fallos y latencias a través de la red.

### 2.2.8 FaaS

[11] Un nuevo tipo de computación *serverless* son las funciónes como servicio o *FaaS* que es un modelo de servicio de computación *Cloud* que esta alterando la manera que las aplicaciones y los sistemas se construyen. A diferencia del *serverless FaaS* permite ser desplegado sobre clústeres de Kubernetes independientes de la *Cloud* eliminando esa posible dependencia del proveedor.

En *FaaS,* la lógica del lado del servidor se mantiene en la responsabilidad del desarrollador, como en arquitecturas tradicionales, pero la lógica del lado del servidor se ejecuta en contenedores de computación sin estado. Estos contenedores se ejecutan bajo eventos, normalmente efímeros y gestionados complemente por un proveedor externo.

Como cualquier otro servicio ofrecido *FaaS* es una plataforma de terceros que solo pagas por uso. Esta plataforma permite centrarse en el diseño, ejecutar y gestionar funcionalidades de la aplicación olvidándose de la infraestructura *backend.* Es precisamente esta razón por la que se está empezando a adoptar *FaaS*, para centrarse en el producto final y menos en las configuraciones de diseño de aplicaciones tradicionales.

Otro factor añadido es que no es necesario utilizar un framework o librería específica, ya que el servicio ya sabe manejarlo. Los despliegues en *FaaS* son diferentes a los de arquitecutras tradicionales en vez de utilizar servidores de aplicación, el código de *frontend* se sube al servicio y el proveedor se encarga de todo el provisionamiento.

La diferencia de los proyectos *FaaS* es que cualquier proceso puede convertirse en una función *serverless* mediante el componente *watchdog* en un contenedor Docker. Esto significa que:

- Se puede ejecutar código en el lenguaje que quieras
- Por cuando tiempo sea necesario
- Donde quieras

El componente del *watchdog,* que se puede ver en la *Figura 11,* proporciona una capa genérica y sin gestión por encima de las funciones. Su función es de recibir las peticiones HTTP aceptadas en el API *Gateway* o gestionar otros eventos que puedan acabar ejecutando una de las funciones.

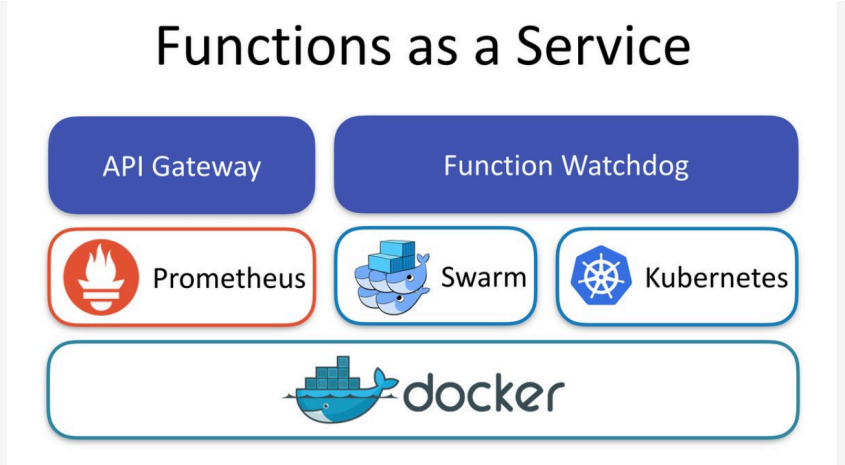



**Figura 11: [12] FaaS stack**

### 2.2.9 Soluciones disponibles

[9] Hoy por how, la implementación más popular de *FaaS* es AWS Lambda principalmente por ser la primera en surgir. Esto conlleva a que es la que tiene un nivel de innovación mayor. Aún así, están apareciendo distintos proveedores que evolucionan sus soluciones y surgen como alternativas a las funciones Lambda:

- GCP o Google Cloud Functions
- Azure Functions
- IBM Apache OpenWhisk Function
- Alibaba Cloud Function Compute
- Auth0 Webtask
- Spotinst Functions

También aparecen opciones que permiten abstraerse del proveedor con herramientas de despliega independientes de la infraestructura utilizada:

- Zeit, la plataforma *serverless* global que ofrece tanto un modelo de despliegue abstracto como infraestructura sobre AWS, Google Cloud o Azure.

Las soluciones *open source* pueden ser desplegadas en entornos *on-premise* o sobre máquinas virtuales o clústeres de Kubernetes:

- OpenFaas
- Fission
- Kubeless

También aparecen *frameworks* que permiten abstraer de la solución elegida con lo que conlleva una ventaja adicional para poder desplegar según sea conveniente:

- *Serverless Framework*, la herramienta más adoptada para construir aplicaciones *serverless*.
- *Pulumi,* modelo de programación nativo *Cloud* para aplicaciones *serverless* centrado en orfrecer un *framework* de abstracción sobre los distintos servicios *Cloud*.

## 2.3 IoT

[3] El internet de las cosas o *IoT (Internet of Things)* encapsula una visión en un mundo en el que billones de objetos con inteligencia embebida, medios de comunicación, sensores y capacidades de actuación conectadas sobre redes IP.

El *IoT* llama a entornos abiertos y arquitecturas integradas con plataformas interoperables. Los objetos *Smart y sistemas* ciberfísicos o, simplemente "*things"* (concepto que se usará de aquí en adelante) son las nuevas entidades *IoT.* Entre ellos, los objetos cotidianos, con microcontroladores, transceptores ópticos o radio, sensores, actuadores, y un stack de protocolos adecuado para la comunicación en entornos limitados por los recursos disponibles de *hardware*, permitiendo que recolecten datos del entorno y actuar en base a ello, dándoles una interfaz al mundo físico. Están altamentente limitados por memoria, energía almacenada y requisitos de bajo coste. El almacenamiento de datos, procesos, y analíticas son requisitos fundamentales, necesarios para enriquecer los datos en brutos *IoT* y transformarlo en información útil.

De acuerdo con el paradigma de *Edge Computing*, la introducción de recursos de computación en el *edge* permite tener diversos beneficios que son claves para los escenarios *IoT*:

- Baja latencia
- Capacidades en tiempo real
- Conscientes del contexto

Los nodos *edge* pueden actuar como una interfaz de stream de datos que proviene de dispositivos conectados, objetos y aplicaciones. El almacenamiento *Big Data* puede ser procesado con nuevos mecanismos, como machine y Deep learning, transformando datos en bruto generados por los objectos conectados en información útil.

El *IoT* y sus aplicaciones se vuelven una realidad que está empezando a aparecer en proyectos como Smart Cities o Smart Home. Aunque, debido a la velocidadcon las que se quiere implementar se están produciendo soluciones verticales cerradas. Esto está en contra de la como debería ser el *IoT,* un conjunto de soluciones compatibles y entornos interoporables.

Las aplicaciones *IoT* actuales consideran los dispositivos personales como un puente entre los objetos inteligentes y el usuario. Típicamente, los usuarios han de ejecutar *software* en sus dispositivos para interactuar con objetos *IoT*. Estas aplicaciones han de saber a priori con que *things* tienen que trabajar estableciendo protocolos de comunicación, formato de datos y patrones de interacción específicos de la aplicación. A largo plazo esto implica una barrera al progreso por diversas razones:

- Objectos inteligentes manufacturados por el mismo proveedor típicamente han de ser accedidos por *software legacy*, resultando en una cantidad ingente de aplicaciones que los usuarios tienen que instalar para utilizarlo. Esto implica un impacto a la hora de crear nuevas aplicaciones para objetos inteligentes.

- La movilidad es un factor crítico ya que las personas acceden a los servicios y usan las aplicaciones activamente. Esto implica que pueden entrar en entornos con objetos inteligentes que no se han cruzado anteriormente. Aunque hay estándares de mecanismos de configuración para descubrimiento de servicios y recursos estos enfoques no son adecuados si un usuario quiere interactuar completa y perfectamente con los *things.*

- Los objetos inteligentes deberían poder adaptarse dinámicamente a determinadas condiciones, como cambios de nivel de batería, recursos o la presencia de determinados usuarios. Los objetos inteligentes deberían poder tener diferentes formas de interacción, las cuales han podido no ser tenidas en cuenta durante la implementación del *software* y por lo tanto requiere actualizaciones.

### 2.3.1 Arquitectura *IoT* basada en IP

[3] *IoT* requiere interacciones adaptativas entre humanos y objetos inteligentes, con el objetivo de rellenar el vacío entre los usuarios, el mundo físico y el mundo digital. Todos los involucrados han de utilizar el mismo lenguaje, es decir, IP. Para evitar la fragmentación y que aparezcan soluciones verticales se han realizado una serie de objetivos entre equipos de investigación y organizaciones estándar como IEEE, IETF e IPSO. Estos son:

- Definir estándares abiertos para la comunicación

- Mapear el stack tradicional basado en IP de internet al de *IoT.*

Por lo tanto, el *IoT* será una red heterogénea de dispositivos interconectados. Esto será la infraestructura de lo que se conoce como "*Web of Things"* (*WoT*). Aún hay bastante trabajo para hacer para que los usuarios finales puedan acceder y explotar el *IoT* con la misma simplicidad que experimentan en la web. En este punto, es hora de comenzar realmente a interactuar con objetos inteligentes, no solo de comunicarse con ellos.

La *WoT* está siendo diseñada alrededor de los ya conocidos conceptos y prácticas derivados de la Web, como el paradigma *REST.* Este paradigma fue introducido para desacoplar las aplicaciones cliente con los sistemas con los que interactúan y proporcionar robustez. Aunque esto se demuestra que encaja perfectamente en estenarios *maching-to-machine* (*M2M*), el desacoplo introducido en la capa de aplicación no es suficiente para habilitar la adopción de aplicaciones que requieren a humanos estar en el ciclo.

La evolución hacia el *IoT* comienza por repensar y optimizar todas las capas relevantes de la pila de protocolos. A continuación, se describe las características principales de los protocolos de comunicación que se consideran para el *IoT*. En la Figura 12 se pueden observar el modelo *OSI* clásico y el modelo *TCP/IP*, para este último se describirán las distintas capas en *IoT*.

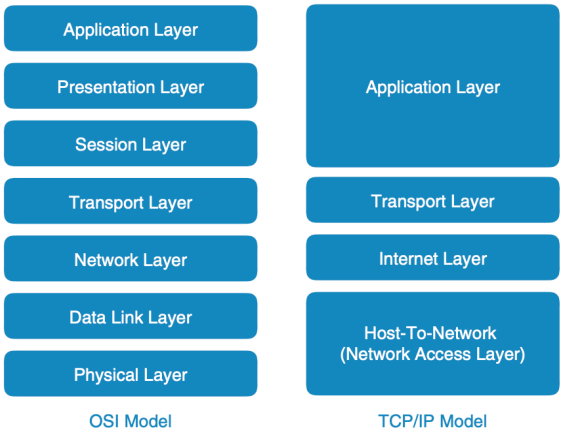



**Figura 12: [3] Modelo OSI**

### 2.3.1.1 Capa física y enlace

[3] Hay cuatro grupos protocolos de comunicación relevantes en los escenarios *IoT*:

- IEEE 802.15.4 and ZigBee, en el estándar se define la capa física y la subcapa de acceso al medio (*MAC*) para conectividad *WPAN* de baja complejidad, bajo consumo de potencia y baja tasa de bit. En base a este estándar se define ZigBee que explota la cooperación para permitir el intercambio de mensajes multi-salto y su red lógica, seguridad y gestión de funciones de aplicación sobre el estándar que define las capas superiores del pila de protocolos

- Low Power WiFi o IEEE 802.11ah, una de las características bajo consumo de energía que permite usarlo en una variedad de nuevos casos como hogares inteligentes, automóviles conectados y atención médica digital, así como en entornos industriales, minoristas, agrícolas y de ciudades inteligentes. Incluye nuevas capas física y MAC, agrupando dispositivos en mapas de indicación de tráfico para acomodar unidades pequeñas (como sensores) y comunicaciones M2M.

- Bluetooth and BLE, es un protocolo de comunicaciones estándar diseñado principalmente para bajo consumo de energía y cortos rangos de comunicación. BLE es un subconjunto de Bluetooth 4.0 con una nueva pila de protocolos para la creación rápida de enlaces simples. Dos características que lo diferencian son:

    - Mayor eficiencia en términos de consumo de energía y la relación de energía de transmisión por bit transmitido que ZigBee.

    - Permite que los dispositivos funcionen como maestros o esclavos en una topología en estrella.

- PLC, implica el usio de los cables electricos existentes para transportar los datos. Representa un área de aplicación interesante para tecnologías *IoT*.

#### 2.3.1.2 Capa de red

[3] El *IoT* basado en IP previsto consistirá en trillones de dispositivos conectados por lo que está magnitud esta magnitud sin precedentes exige que se tomen decisiones estratégicas en el diseño de una red global de objetos inteligente. Por un lado, existe la necesidad de abordar cada objeto inteligente de forma individual y global. Por otro lado, el uso de IPv4 no puede ser un enfoque a largo plazo.

Las direcciones IPv4 escasearían hacienda imposible asignar nuevas IPs teniendo que introducir técnicas *NAT* para proporcionar un espacio de direcciones extendido a los objetos inteligentes. Esto implicaría configuraciones complejas que pondría en riesgo la escalabilidad, capacidad de gestión y facilidad de despliegue de objetos inteligentes.

Partiendo estas premisas, la única solución lógica es adoptar IPv6 en la capa de red. Esto aportaría una serie de características beneficiosas que lo hacen conveniente:

- sus direcciones tienen una longitud de 128 bits, lo que permite asignar aproximadamente 3,4 × 1038 direcciones IP únicas

- integra IPSec para security

- proporciona direcciones locales del enlace y globales, derivadas de la dirección *MAC* del dispositivo.

Aparece una capa de adaptación *6LoWPAN* que es la especificación de mapeo de los servicios necesarios para mantener una red IPv6 a través de *WPANs* de baja potencia. Este estándar proporciona cabeceras de compresión para reducir las cabeceras de transmisión, fragmentándolas para poder cumplir con los requisitos de *MTU* de IPv6 y el reenvío a la capa de enlace para soportar la entrega de múltiples saltos. Es decir, poder transmitir un pequeño datagrama de IPv6 un único salto de IEEE 802.15.4.

*LoWPAN* se caracteriza por:

- **Tamaño de paquetes pequeños**, el máximo tamaño de paquete de capa física es 128 byts.

- **Bajo ancho de banda**, no permite enviar los datos con tasas de transmisión altas.

- **Baja potencia**, algunos dispositivos operan con batería por lo que el consumo de energía es un factor crítico.

- **Bajo coste**, los dispositivos típicamente son sensores, *switches* o similares de bajo coste.

- **Poco fiable**, son sujetos a incertidumbre por conectividad radio, se acaben las baterías.

- **Ciclos de trabajo**, los dispositivos pueden estar "durmiendo" por largos periodos de tiempo para ahorrar energía, por lo que no están disponibles para comunicarse durante estos periodos.

#### 2.3.1.3 Capa de transporte

[3] Los escenarios *IoT* típicamente exigen aproximaciones de comunicación eficientes de energía, ligeras y no intensivas en *CPU*. Esto se debe a que los objetos inteligentes tienen capacacidades límitadas. Por esa misma razón, *UDP* es la opción típica para la comunicación en la capa de transporte *IoT*. Esto conlleva a que se pierdan las características de *TCP* de transmisiones, ordenado y control de congestión que han de ser implementadas a una capa superior por la aplicación.

#### 2.3.1.4 Capa de aplicación

[3] La experiencia de Internet y la importancia atribuida a los protocolos de la capa de aplicación ha sido crítica para la conciencia de que *IoT* necesita una capa de aplicación dedicada, que debe tener en cuenta los requisitos de las capas inferiores. Por lo tanto, se pueden definir tres protocolos principales de capa de aplicación:

- ***HTTP***:

El protocolo *HTTP* es el protocol de capa de aplicación más popular, siendo un protocolo de petición y respuesta basado en texto sin estado que define la comunicación entre el cliente y servidor sobre conexión *TCP*. Se caracteríza por:

- Se utilizan *URLs* para dar direccionar a los recursos que reciben peticiones.
- Se utilizan métodos para definir el tipo de operación ejecutada por el servidor, los principales métodos son GET, POST, PUT, y DELETE.
- Se pueden añadir cabeceras en los mensajes para dar más información y, por lo tanto, puedan ser procesados correctamente además de proporcionar semántica adicional para la comunicación.
- Códigos de estado en las respuestas que proporcionan un estándar, uniforme y descriptivo para informar al cliente sobre el resultado de una petición.

- ***CoAP***

*CoAP* ha sido diseñada para traer el paradigma *REST* a *IoT* implementando un modelo de comunicación petición/respuesta sobre *UDP* soportando cuatro métodos básicos POST, GET, PUT y DELETE.

Está diseñado para ser usado con dispositivos limitados en terminados de capacidades computacionales, que pueden estar caracterizados por una batería limitada y operar en redes limitadas. Siendo un protocolo *RESTful* en el que las *URIs* identifican el recurso de la aplicación. La representación del recurso es el actual o estado deseado de un recurso referido por un *namespace*.

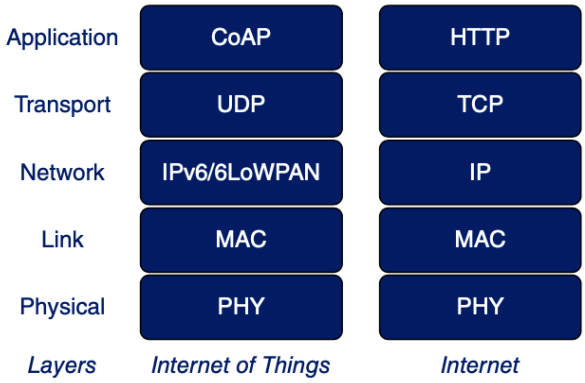



**Figura 13: [3] *CoAP* vs *HTTP***

*CoAP* mapea con *HTTP,* de una manera sencilla para asegurar una integración completa con la web. Esto permite realizar traducciones del protocolo de manera sencilla en proxis dedicados para garantizar una interoperabilidad completa entre los clientes *HTTP* y servidores *CoAP*, y viceversa.

- ***MQTT***

*MQTT* es un protocolo ligero de publicación y subscripción basado en TCP. Este protocol está enfocado en entornos en el que los dispositivos y redes están limitados. Se puede utilizar en escenarios en el que la comunicación de dos vías entre *endpoints* operando en redes no fiables. La naturaleza ligera de *MQTT* lo hace bastante compatible con entornos limitados en el que las cabeceras de los mensajes y el tamaño de los mensajes han de ser mínimos.

Según este modelo, los mensajes *MQTT* se publican en un espacio compartido dentro de un bróker. Los *topics* son utilizados como filtros de *streaming* de mensajes de todos los publicadores al bróker. Los mensajes se envían a todos los clientes que se han subscrito a un filtro de *topics* emparejado. Esto implica que un único cliente puede recibir múltiples mensajes de múltiples publicadores. Es posible subscribirse solo a una porción de un *topic* debido a la jerarquía. También se pueden emplear *wildcards* o comodines para proporcionar mayor granularidad sobre el mensaje recibido.

En *MQTT,* el *bróker* aplica los filtros de subscripción al *stream* de mensajes recibidos para determinar eficientemente a que cliente debería entregar el mensaje. Por lo tanto, una subscripción se puede considerar como una operación condicional en tiempo real y no tiene naturaleza de larga duración.

### 2.3.2 Aplicaciones *IoT*

[3] Los objetos inteligentes, especialmente los que están basados en baterías, no pueden procesar una cantidad muy pesada de procesos de carga y usar protocolos de comunicación intensos. Esto se debe a los requisitos funcionales y económicos costes bajos, limites de tamaño y consumo de energía mínimo que son una de las razones por la que los dispositivos *IoT* tienen límites de capacidades computacionales. Por un lado, Limitar las capacidades de procesos significa que es difícil procesar mensajes grandes.

Por el otro, menos procesamiento significa menos consumo de energía. Como consecuencia, los dispositivos *IoT* necesitan minimizar la cantidad de datos transmitidos.

Los dipositivos que operan en redes de baja potencia y con perdidas o *LLNs* (*Low-power and Lossy Networks*), se pueden beneficiar enormemente de protocolos ligeros. Mensajes grandes implican más fragmentación, que introducen sobre cabeceras que debido a la naturaleza inestable de *LLNs*, trasmitir más fragmentos requiere múltiples retransmisiones hasta que todo el mensaje sea transmitido satisfactoriamente. Estas retransmisiones conllevan retardos y mayor consumo de energía.

Los protocolos de comunicación son paradigmas de comunicación específicos que se pueden categorizar en petición/respuesta y publicación/subscripción. Cada escenario de una aplicación requiere la elección del paradigma de comunicación y protocolo óptimo. No hay una definición de arquitectura que encaje en todos los escenarios.

### 2.3.3 *Cloud Computing IoT*

[14] Una plataforma *Cloud IoT* es donde las capacidades de las pilas tecnológicas *IoT* y la computación *Cloud* se unen para proporcionar un valorar añadido a los consumidores y las aplicaciones del negocio. En el esfuerzo por poner online objetos físicos y hacer que se comuniquen, cooperen y actúen de manera inteligente sin la intervención humana, hace que *IoT* de penda de plataformas *IoT* que permita aprovisionamiento, gestión y automatización de objetos inteligentes dentro de una infraestructura *IoT.*

Los enornos *IoT* son una combinación de tecnologías de distintos proveedores que componen un ecosistema complejo y heredado el cual, sin una base común de integración, estaría fragmentado y, por lo tanto, no podría funcionar. Por ello, se puede decir que las plataformas *IoT* proporcionan un punto común para todos los dispositivos interconectados y sirve para recolectar y manejar los datos que transmiten sobre la red.

Actualmente, en el caso de muchas empresas que tienden a *IoT*, tienen una elección clara de establecer una infraestructura de servidores onpremise costosa, vulnerable y difícil de escalar o una solución *IoT* basada en *Cloud* que permite evitar costes innecesarios sin presentar ninguna perdida considerablemente en cuando a funcionalidad de la plataforma. Se verán las distintas ventajas y desventajas en los siguientes apartados 2.3.4 y 2.3.5 respectivamente.

[15] La combinación de *IoT* y *Cloud* conducirá inevitablemente al desarrollo de la economía. Si una combinación de *IoT* y *Cloud* es descrita periódicamente se puede distribuir en las siguientes tres fases:

- En la primera fase, **la información y los datos se toman como factores a utilizar**, lo que mejora la eficiencia de las empresas.

A través de la combinación de *Cloud* con *IoT* se puede obtener más datos e información útil. La enorme capacidad de computación y almacenamiento de la *Cloud* puede reflejar el estado de los procesos productivos y hacer frente a varias necesidades. Toda la información y los datos se pueden centralizar y gestionar. Esto conlleva la a posibilidad de hacer un análisis detallado de los datos que permitirá llegar a conclusiones, obtener

resúmenes, hacer ajustes continuamente y optimizaciones de todo el sistema haciéndolo altamente eficiente y de bajo coste.

- La segunda fase, **cambia la forma tradicional de producción y gestión**, optimizando gradualmente la estructura del sistema

El foco es la optimización de los modelos de producción basados en la gestión y uso de la información. Tecnologías *IoT* inteligentes implementan monitorizaciones automáticas y programaciones remotas de los procesos de producción, haciendo que la producción consiga automatización y sea centralizada. Además, mediante las plataformas inteligentes de información de la *Cloud*, se puede reducir los errores humanos consiguiendo un control de precios y patrones de gestión.

- La tercera fase, **impulsa la fusión de la información y optimiza la estructura** de la industria en su conjunto.

Está fase se fusiona la información y se promueve la completa reconstrucción o actualización de los negocios. Permitiendo que *IoT* consiga innovación continua en el modelo de servicio y la tecnología *Cloud* construye una plataforma colaborativa de modo de producción auxiliar. Nuevas evoluciones y fusiones de información permiten crear y actualizar nuevas tecnologías que aporten valor añadido.

### 2.3.4 Ventajas de *Cloud Computing* para *IoT*

[14]

- **Escalabilidad**, en un sistema *IoT* basado en *Cloud* añadir nuevos recursos es automatizable y rápido adaptándose a las necesidades. Permite mayor flexibilidad en caso de querer limitar los requisitos de almacenamiento o reducir el número de dispositivos *IoT.*

- **Mobilidad de datos**, los datos en la *Cloud* son accesibles desde cualquier parte del mundo sin limitaciones de infraestructura o red. Además, una plataforma *IoT Cloud* te da todas las herramientas para provisionar, gestionar y actualizar los dispositivos, sensores y procesos que adquieren data remotamente en tiempo real.

- ***Time to Market,*** con soluciones *IoT Cloud* lleva menos tiempo y esfuerzo implementarlas y reduce significativamente los costes. Infraestructuras *IoT* basadas en *Cloud* acaban siendo más rentables cuando el *time to market* es un factor crucial para el negocio.

- **Seguridad**, debido a las actualizaciones de *software* y *firmware* regulares y una monitorización completa ofrecida por algunos proveedoraes *Cloud* va reduciendo las brechas de seguridad. Aún así, se puede debatir si tener datos sensibles controlados por terceros implica un problema de seguridad.

- **Efectividad de costes**, se elimina la necesidad de realizar inversiones de antemano además de evitar los mantenimientos necesarios de *hardware*. En base al modelo de negocio los costes son sencillos de predecir al estar basados en pago por uso, además de evitar posibles perdidas debido a caídas de los sistemas.

### 2.3.5 Desventajas de *Cloud Computing para IoT*

[3] Inicialmente *IoT* se caracterizada por la adopción de una aproximación muy simple de interconexión de dispositivos dependiendo de la disponibilidad de los servicios *Cloud*, simplemente había que conectarse a internet y subir todos los datos a la *Cloud.* Mediante el cual los servicios *Cloud* proporcionan servicios de almacenamiento de todos tus dispositivos en un mismo sitio y, por otro lado, una interfaz basada *HTTP* para ser accedida por los clientes.

Actualmente los proveedores *Cloud* han comenzado a proporcionar sus propias plataformas *IoT.* Estás plataformas son una forma sencilla de desplegar las aplicaciones sin necesitar la inversión del desarrollo de los recursos de *backend*. A pesar de que es una implementación muy efectiva en cuanto a costes, se ha dado el malentendido de que el *IoT* simplemente se puede construir conectando las cosas a Internet. Siendo este uno de los requisitos, pero no suficiente pare crear una red de interconexión de dispositivos. Este *hardware* y *software* de primera generación ha introducido varios problemas debido a no controlar la red ni tenerla en cuenta:

- **Escalabilidad**, el número de dispositivos *IoT* total se espera que incremente enormemente. No se sabe si actualmente serán capaces de gestionar los servicios y redes todo el tráfico generado por millones de *things* a largo plazo.

- **Disponibilidad**, si la conexión no está disponible temporalmente puede causar problemas graves y depender de la *Cloud* puede causar que esto ocurra.

- **Interoperabilidad**, todas las aplicaciones de dispositivos a la *Cloud* no permiten la interacción entre *things* desarrollados por diferentes fabricantes.

- Las **soluciones basadas en *Cloud*** nunca van a desaparecer, aunque este tipo de arquitectura no debe ser la única referencia de un *IoT* que es escalable y está en evolución.

En definitiva, hay que valorar para que casos las soluciones *IoT* basadas en *Cloud* pueden ser óptimas. Sobretodo hasta que punto han de depender de la *Cloud* y que alternativas pueden surgir

### 2.3.6 *Edge* y *Fog Computing*

[15] *Fog computing* o computación de niebla*,* también se podría conocer como “Edge Computing” (pero con unas pequeñas diferencias que se verán más adelante), es un paradigma nuevo que fue introducido para permitir soportar requisitos como soporte de movilidad, baja latencia y conciencia de ubicación.

El objetivo principal es mover algunas de las computaciones basadas en la *Cloud* y almacenamiento en el *edge* de la red. El *Fog* acerca a los usuarios funcionalidades que mejoran el rendimiento, permitiendo el desarrollo de nuevas aplicaciones que necesiten

consumos de baja latencia y la necesidad de conocer información relacionada con la ubicación.

Las características de *Fog computing* son las siguientes:

- Amplia distribución geográfica, en contraste con la visión centralizada de la *Cloud*
- Modelo de subscripción usado por los miembros del *fog*
- Soporte y movilidad

Conlleva un nuevo enfoque en las redes de acceso a internet haciendo que la computación, almacenamiento y recursos de red estén disponibles en el *edge* de las redes de acceso. Esto mejora el rendimiento, minimiza latencias y maximiza la disponibilidad, ya que los recursos son accesibles, aunque el acceso a internet no lo esté.

Este tipo de soluciones buscan introducir una nueva capa arquitectural intermedia en que los recursos y la aplicación están disponibles gracias a la proximidad con los dispositivos, evitando la necesidad de acceder continuamente a la *Cloud.* Mientras que arquitecturas únicamente *Cloud proporcionan* soluciones escalables y flexibles distribuyendo los recursos en servidores, tiene una serie de desventajas en parte ya mencionadas:

- Latencia
- Disponibilidad y dependencia de conectividad a Internet para realizar operaciones.
- *QoS* y *QoE*
- Seguridad y privacidad

Debido a los beneficios sobre arquitecturas basadas en *Cloud*, especialmente si los tiempos son críticos o la conectividad a Internet es pobre o ausente, *Fog computing* plantea tener un papel clave en el despliegue de aplicaciones *IoT*. Aunque, no está pensada para reemplazar la *Cloud*, si no complementarla.
Las redes de acceso basadas en *Fog* están ideadas con la presencia de nodos altamente especializados, conocidos como nodos *Fog,* que permiten ejecutar aplicaciones distribuidas en el *edge* de la red.

Este tipo de arquitectura puede proteger los recursos proporcionando acceso remoto a las replicas de una manera transparente. Donde los recursos locales se mantienen sincronizados

Estos nodos *Fog* pueden estar compuestos por máquinas virtuales o incluso contenedores. Los contenedores permiten entornos flexibles y están ganando una atención considerable para *Fog computing.* Cambiando de un paradigma descentralizado permite habilitar procesos que son descargados en el *edge*, reduciendo los tiempos de respuesta y por lo tanto una mejora de experiencia de usuario. Si además para este nuevo paradigma aplicamos el concepto *serverless* podría proporcionar aún mejores resultados, conocido como *Fog function.*

Aunque este apartado se ha centrado más en *Fog computing* es necesario compararlo con *Edge computing* esto se verá en el siguiente apartado, donde se podrán ver las diferencias principales.

Edge computing es una filosofía de red centrada en traer la computación lo más cerca al origen de datos que sea posible para reducir latencias y ancho de banda. Es decir, ejecutar menos procesos de computación en la *Cloud* y moviendo esos procesos en dispositivos locales.

#### 2.3.6.1 Diferencias

[17] Tanto *Fog computing* como *Edge computing* proporcionan las mismas funcionalidades en cuanto a acercar los datos y la inteligencia en plataformas analíticas que pueden están en o cerca del origen de datos. Se puede decir que conceptualmente ambas cosas son de manera efectiva lo mismo, ambas tienen el objetivo de realizar tareas de computación que normalmente se harían en la nube.

Ambas tecnologías ayudan a reducir la dependencia de plataformas basadas en *Cloud* para analizar datos, que pueden producir problemas de latencia, y en cambio es posible hacer decisiones basadas en datos con mayor rapidez. La diferencia principal es donde se realiza el procesado de los datos.

*Edge computing* normalmente se ejecuta directamente en los dispositivos en los que el sensor está conectado o actúa como *gateway* (puerta) que está cerca de los sensores. Sin embargo, *Fog computing* mueve las actividades de computación *edge* a procesadores que están conectados a la *LAN* o en el *hardware LAN* por lo que están físicamente más distante de los sensores y actuadores.

Por lo tanto, con *Fog computing*, los datos se procesan en un nodo *fog* o *gateway IoT* que está situado dentro de la *LAN.* En cambio, para *Edge computing*, los datos son procesados en dispostivos o sensores sin tener que transferirlos a otro lado.

#### 2.3.6.2 Ventajas y desventajas de *Edge Computing*

- **Ventajas**

Proporcionan mayor seguridad, ya que los datos no se transfieren y por lo tanto son más seguros. Mantiene todos los datos y procesos en el dispositivo en el que se crearon. Esto mantiene los datos discretos y contenidos dentro de la fuente de la verdad, el dispositivo originario.

Además, se ahorra tiempo y recursos de operaciones de mantenimiento recolectando y analizando datos en tiempo real. Las redes en el *edge* proporcionan analítica casi en tiempo real que permite optimizar el rendimiento y la disponibilidad.

- **Desventajas**

Un factor relevante es no saber que hacer con los datos y por lo tanto, saber cuando es necesario enviar los datos a la *Cloud* y cuando es mejor procesarlos en local. En algunos

casos es necesario que se lleven a la *Cloud* para procesar un determinado conjunto de datos que no se puede procesar en el *edge* por su complejidad.

#### *2.3.6.3* Ventajas y desventajas de *Fog Computing*

- **Ventajas**

Con el almacenamiento de datos y procesado que están en una arquitectura *Fog computing* permite a las organizaciones agregar datos de multiples dispositivos en almacenes regionales.

Por otra parte, mientras *Fog computing* permite recolectar datos de múltiples dispositivos tiene una mayor capacidad para procesar datos, mejorando las capacidades *edge.* Siendo conveniente implementarlo cuando se tiene millones de dispositivos compartiendo datos constantemente.

- **Desventajas**

El dimensionamiento es un posible inconveniente ya que a mayor infraestructura hay que realizar una mayor inversión y dependes de la constancia a través de una red amplia.

## 2.4 Resumen

El origen de *Cloud Computing* aparece por la necesidad de reducir la inversión en recursos de computación propios (*on premise)*. Estos recursos de computación han de tener una planificación previa en cuanto a su dimensionamiento. Por ello, surge la posibilidad de que los recursos de computación estén infrautilizados y, además, al ser recursos limitados por su dimensionamiento no es posible anteponerse a las necesidades de escalabilidad o picos de computación que puedan aparecer.

Todo esto implica una gran inversión de capital (*CapEx*) [1] además de todos los costes implicados para el mantenimiento y gestión de estos recursos de computación. Si, en vez tener que invertir de antemano en los recursos de computación, se pudiese pagar solo por lo que se necesita implicaría solo una inversión de operación (*OpEx*). Aquí es donde aparece el concepto de *Cloud Computing* que permite utilizar un modelo de pago por uso o *pay as you use*.

Aquí es donde entra el papel de *serverless,* donde una de las mejores definiciones de una arquitectura serverless se puede encontrar destacando la siguiente cita [9]:

“En este modelo, la computación pasa finalmente a ser como la luz o el agua, una ‘utility’ que consumiremos en función de nuestras necesidades”

Esta definición deja claro las ventajas y características de las arquitecturas *serverless* y, por lo tanto, el modelo de servicio *FaaS.* El término *serverless* [9] es el modelo de computación según el cual el proveedor de la capa de computación nos permite ejecutar durante un periodo de tiempo determinado, cuando un evento lo dispara, porciones de

código denominadas “funciones” sin necesidad de hacernos cargo de la infraestructura subyacente que se provisiona para dar el servicio.

Se puede ver que en este modelo el proveedor se encarga de ofrecer los recursos de forma transparente, de escalarlos automáticamente si crece la demanda y de liberarlos cuando no son utilizados. Por lo tanto, se definen una serie de restricciones de procesamiento y un modelo de pago por el consumo de recursos por ejecución. Típicamente una función no debería ejecutarse durante más de 5 o 10 minutos por ejecución.

Este tipo de modelo de servicio puede encajar con las soluciones *IoT* debido a que llaman a entornos abiertos y arquitecturas integradas con plataformas interoperables. Se encargan de recolectar datos del entorno y actuar en base a ello, dándoles una interfaz al mundo físico.

Pero, están altamentente limitados por memoria, energía almacenada y requisitos de bajo coste. El almacenamiento de datos, procesos, y analíticas son requisitos fundamentales, necesarios para enriquecer los datos en brutos *IoT* y transformarlo en información útil. Esto es un factor clave que puede ayudar a mejorarlo mediante una arquitectura *FaaS.*

De acuerdo con el paradigma de *Edge Computing*, la introducción de recursos de computación en el *edge* permite tener diversos beneficios que son claves para los escenarios *IoT* con baja latencia, capacidades en tiempo real y conscientes del contexto

Los nodos *edge* pueden actuar como una interfaz de stream de datos que proviene de dispositivos conectados, objetos y aplicaciones. El almacenamiento *Big Data* puede ser procesado con nuevos mecanismos, como machine y Deep learning, transformando datos en bruto generados por los objetos conectados en información útil. Todo ello apoyado de las ventajas que proporciona *FaaS* puede ser un factor determinante en las soluciones *IoT.* Esto se verá con mayor detalle en el siguiente apartado.

# 3 IoT y FaaS

El objetivo principal de este apartado es entender porque dentro de los modelos de servicio disponible *FaaS* es con el que se va a explorar. Las soluciones *IoT* se pueden ver beneficiadas debido a las cualidades innatas de las *FaaS* como *serverless*. Esto conlleva un nivel de abstracción que permite un despliegue, gestión y escalado de las soluciones *IoT* independientemente del nodo de computación *edge* y los dispositivos empleados.

Debido a la disrupción de las tecnologías y su gran velocidad de desarrollo aparece una necesidad de poder implementar soluciones escalables, flexibles, estables y con costes bajos para las soluciones *IoT*. Por lo tanto, el concepto de arquitectura *serverless* encaja perfectamente con estas necesidades. Pudiendo conseguir desplegar soluciones *IoT* sin la necesidad de tener una infraestructura, ni costes de mantenimiento y prácticamente ningún coste de operación.

En concreto, hay que considerar las *FaaS* como una de esas nuevas tecnologías emergente y disruptivas que pueden cambiar el modelo de arquitectura de las

aplicaciones y soluciones. Por lo tanto, estudiar como encaja en una solución *IoT* para ver si la transición a este nuevo paradigma es beneficiosa y la complejidad que conlleva.

## 3.1 Por qué es relevante una arquitectura *serverless* para las IoT

[16] *IoT* es una de las soluciones ya se ha adaptado despliegues *serverless,* hay diversas razones para desplegar *IoT serverless* entre las que se incluyen:

- Al construir un *backend* basado en *Cloud* escalable para monitorización *IoT* y control, problemas de escalabilidad y disponibilidad pueden ser díficiles de resolver. Las redes *IoT* suelen estar compuestes de millones de nodos, a través de distintas zonas geográficas, utilizando una arquitectura *serverless* permite eliminar toda la gestión sobre los proveedores *Cloud*.

- Las redes *IoT* suelen tener variaciones de carga donde a ciertas horas del día o debido a determinados eventos se puede desencadenar que se active un gran número de dispositivos. Debido a que los servicios *IoT* suelen ejecutar un gran número de dispositivos inactivos, estos están esperando a ser activados y, por lo tanto, tener todos los procesos con recursos reservados no es óptimo y causará sobrecostes por infrautilización. Construir estos servicios con una arquitectura *serverless* permite los consumidores pagar por lo que realmente consumen.

- El software que ejecuta y habilita los servicios de IoT generalmente comienza de manera simple, pero se van agregando características continuamente. La adopción de metodologías *serverless* permite a las organizaciones centrarse en su valor comercial principal, agilizar los cambios y el time to market en vez de la orquestación y el mantenimiento.

A pesar de e todas las ventajas que aportan este tipo de arquitecturas y se entienden como un modelo clave para *IoT,* sigue apareciendo el problema de tener todos los procesos de computación el *Cloud* ya mencionados en el apartado 2.3.5. Aquí es donde aparece el concepto de [19] Deviceless Edge Computing que introduce la arquitectura *serverless* y utilizar los dispositivos como servicios en los nodos *edge*.

[19] Recientemente, *Cloud Computing, Edge Computing* e *IoT* ha estado convergiendo cada vez más fuertemente, provocando la creación a gran escala, sistemas distribuidos geográficamente. Estos sistemas explotan los modelos y tecnologías de *Cloud Computing*, principalmente utilizando data centers remotos, pero también con *Cloudlets* para aumentar el rendimiento de dispositivos *edge* limitados por los recursos.

La computación *serverless* es un paradigma emergente, donde la arquitectura de *software* se compone de factores desencadenantes (*triggers*) y acciones, siendo una plataforma que proporciona aislamiento de la infraestructura subyacenete, gestionando y ejecutando funciones *FaaS*, que lo hacen sencillo de desarrollar, gestionar, escalar y operar.

Aunque inicialmente fueron diseñadas para despliegues *Cloud* centralizados, los beneficios del paradigma *serverless* se vuelve especialmente evidente en el contexto de *Edge computing*. Esto se debe a que estos sistemas, infraestructura tradicionales y servidores de aplicaciones son tediosos, inefectivos, propensos a errores y costosos. Por

suerte algunas de las técnicas *serverless*, como la ejecución de espacio aislado del código proporcionado por el inquilino multilenguaje se puede aplicar en *edge* sin modificaciones sustanciales.

Desafortunadamente, debido a la naturaleza diferente de la infraestructura Edge, por ejemplo, en términos de recursos disponibles, red, distribución geográfica, gran escala, etc., supuestos fundamentales de arquitectura y diseño detrás de sistemas sin servidor basado en *Cloud* computación *serverless* ha de ser estudiada y específicamente adaptada a la infraestructura *Edge* para conseguir *Device Edge Computing*. Algunas de los principales desafíos que aparecen incluyen:

- **Agrupación de recursos y elasticidad rápida**, ya que la elasticidad en el *edge* implica retos no presentes en *Cloud* principalmente por la variación de la naturaleza de su infraestructura, las topologías de conectividad de red y conciencia de recursos locales.

- **Seguridad**, el *Edge* requiere mayor protección y aislamiento ya que está más expuesto a ataques.

- **Provisionado automático y gestión a escala**, son necesarias técnicas que puedan interactuar con *Cloud* y *Edge* para este propósito.

- **Programación de recursos *Edge* escasamente desacoplados y escasos**, debido a la naturaleza de los recursos *Edge* no es sencillo conseguirlo.

- **Desarrollo de aplicación *deviceless***, en el paradigma *deviceless* se intercambia gestión de dispositivos explicada por lógicas de negocio de la aplicación. Se ha de considerar los recursos *Edge* involucrados y sus capacidades, pero con nivel de abstracción mayor.

- **Gobierno centrado en *Edge***, sistemas de gobernanza tradicionales han de ser revaluados y particularmente diseñados en el nuevo contexto *Edge.*

Todo esto implica un nuevo paradigma que hay. Aún se deben explorar y analizar tecnologías que permitan conseguir este concepto de arquitectura *serverless.* Se pueden ver un concepto fundamental para poder habilitar este tipo de arquitectura, el dispositivo como servicio que se verá en el siguiente apartado.

Así mismo, existe una tecnología que permite extender las capacidades *Cloud* en los dispositivos locales permitiendo realizar funciones que permitan establecer arquitecturas prácticamente *serverless*, en concreto Greengrass SDK que permite ejecutar funciones *Lambda* e interactuar con los distintos dispositivos en un nodo *edge,* conocido como Greengrass Core. Este será el objetivo de estudio principal que se verá más adelante.

## 3.2 *Device as services*

[18] A medida que el Internet de las cosas (IoT) crece en tamaño, se está volviendo crítico que realicemos operaciones basadas en datos en el *Edge* para reducir la latencia,

el tiempo de respuesta, los costes y el uso de energía para aplicaciones de IoT. Como tal, los sistemas *Edge* cada vez más ubican los datos de servicios de gestión y análisis con detección, en lugar de exigir que los dispositivos envíen sus datos a través de redes de larga distancia para procesamiento remoto utilizando el modelo tradicional de *Cloud.*

Los sistemas *Edge* típicamente implementar un modelo de publicación/subscripción en el que los dispositivos publican *streams* de datos. Se intuye que en un futuro no muy lejano las soluciones *IoT* incluirán dispositivos multi función. En consecuencia, es lo más apropiado un modelo de servicios en el que las especificas funciones pueden ser solicitadas por cada dispositivo cuando sea necesario.

Además, como los dispositivos continuarán siendo recursos limitados, necesitarán servicios complementarios en el *Edge* que aumenten las capacidades y permitan el escalado. Esto implica la aparición del concepto *Devices as Services.*

### 3.2.1 Características

En el paper [19] se describen algunas características de este prototipo:

- Las aplicaciones *IoT* pueden estructurarse como colecciones de servicios que requieren funcionalidades de otros dispositivos, *Edge y* Cloud.

- El procesado de datos en la red puede reducir significativamente el tiempo de respuesta y el consumo de energía.

- El aislamiento *Edge* impide la necesidad de canales de comunicación dedicados entre las aplicaciones y los dispositivos, permitiendo facilitar la protección de la privacidad de datos y dispositivos.

- Actuadores a nivel de dispositivo requieren algún tipo de protocolo de petición/respuesta donde el dispositivo procesa la solicitud

- La heterogeneidad de los dispositivos promueve un paradigma de programación único y un modelo de interacción distribuido

- Dispositivos multi-funcionales pueden y serán capaces de realizar sus funciones basadas en las necesidades de la aplicación.

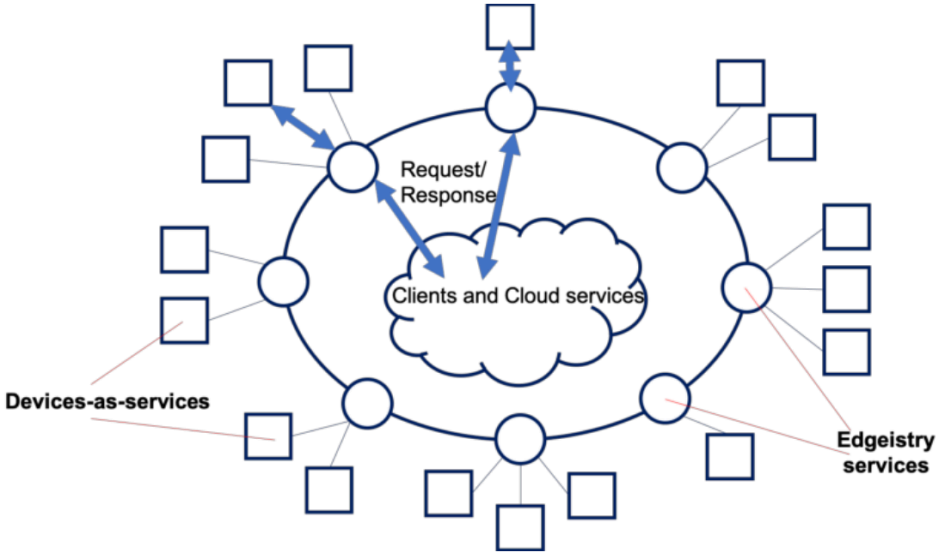



**Figura 14:[19] Modelo distribuido de aplicaciones *IoT***

En el paper [18] proponen un nuevo modelo distribuido para aplicaciones *IoT* donde se da la vuelta al concepto cliente-servidor en el que los dispositivos *edge* son "servidores" que siguen siendo limitados en recursos, atienden múltiples, aplicaciones escalables desplegadas en la *Cloud.* Está arquitectura está basada en programación y modelo de despliegue *FaaS* e integra un nuevo enfoque para el descubrimiento, la autenticación y la autorización eficientes de clientes y servidores a través de una combinación de seguridad basada en capacidades. Además de una serie de servicios *edge* para registrar servicios, mediación de privacidad y optimización llamado *Edgisitry.*

La idea es que la arquitectura *Cloud* y el Internet actual deben acomodarse a aplicaciones *IoT* escalables. En vez de hospedar los servicios en la propia *Cloud* que hace que los dispositivos se vuelvan "servidores" y las aplicaciones ejecutándose en la *Cloud* los clientes.

### 3.2.2 El Edgistry

En el paper [18] utilizacn esta aproximación para separar la provisión de los servicios entre dispositivos y una serie de servicios de gestiones comúnes localizados en el *Edge*, el termino es *The Edgistry.* Los dispositivos hosppedan servicios pequelños y ligeros de recursos que gestionar peticiones desde, y responde a, aplicaciones cliente. Este es el encargado de:

- Implementan un servicio de registro distribuido eventualmente consistente
- Actúa como un servicio de comunicación de coincidencia veloz.
- Protege la privacidad del dispositivo
- Proporciona servicios de distribución de computación y almacenamiento para servicios alojados en dispositivos

Desde una perspectiva de programabilidad, se obtiene una capacidad *FaaS* universalmente portátil. Donde se implementan los servicios en el dispositivo utilizando un micro *FaaS* como una implementación pequeña y especialmente ajustada de *FaaS* dirigida específicamente a microcontroladores con recursos limitados como plataforma de ejecución.

El micro *FaaS* admite las mismas API de programación que una implementación más pesada *serverless* diseñada para ejecutarse en un dispositivo perimetral, en una *Cloud* privada o en una *Cloud* pública. De esta forma, las aplicaciones IoT pueden codificarse utilizando un único conjunto de *FaaS* que permite una programación de abstracciones del dispositivo a las *Cloud*, fomentando el concepto *serverless*. Actualmente existe una *SDK* que permite trabajar bajo este concepto, que se verá a continuación.

## 3.3 AWS *IoT* Greengrass

[20] AWS *IoT Greengrass* es un software que extiende las capacidades *Cloud* a dispositivos locales. Estos dispositivos, pueden recolectar y analizar datos más cerca del origen de la información, actuar de manera autónoma sobre eventos locales y comunicarse de manera segura entre ellos en las redes locales. Los dispositivos locales pueden comunicarse de manera segura con AWS *IoT* Core y exportar datos *IoT* a la *Cloud de* AWS. Los desarrolladores pueden utilizar funciones Lambda de AWS y conectores predefinidos para crear aplicaciones *serverless* que son desplegadas en dispositivos para su ejecución local. Se puede ver la arquitectura básica de Greengrass en la Figura 15.

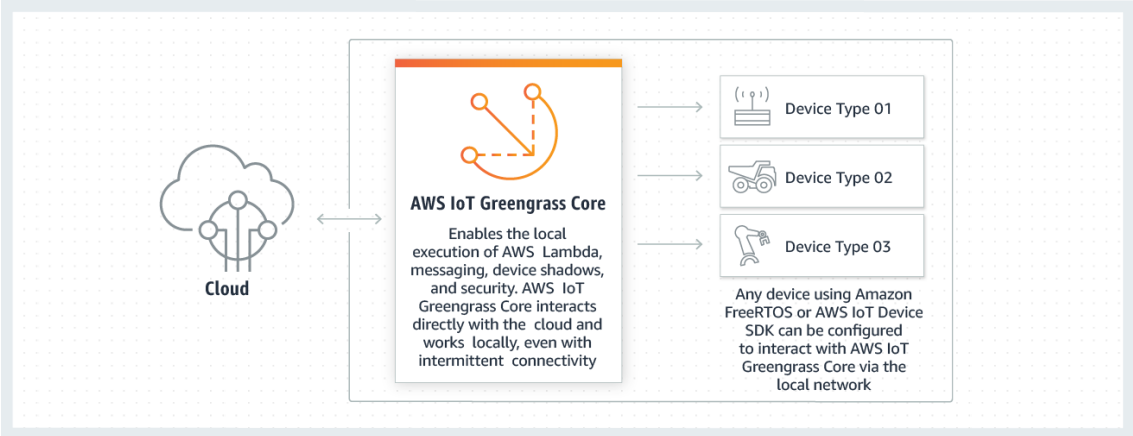



**Figura 15: [20]Arquitectura base de AWS *IoT* Greengrass**

Para conseguir funcionalidades *offline*, AWS expone una capa de *IoT Core* con pequeñas huellas que se ejecutan localmente. AWS *IoT Device* SDK continúa trabajando con Greengrass cuando desaparecen las comunicaciones con la *Cloud*. También expone un registro de dispositivos y un broker de mensajes para dispositivos locales.

Greengrass contiene un entorno de ejecución Lambda que soporta computación bajo eventos. Los desarrolladores pueden escribir el código como funciones Lambda comunes y desplegarlas en AWS Lambda, que después es desplegado desde el panel de control de Greengrass en los dispositivos.

Las funciones Lambda ejecutadas en Greengrass proporcionas los requisitos de comunicación para la capa *Edge computing.* Debido a que las funciones Lambda se ejecutan de manera nativa en el *Edge,* tienen acceso a recursos locales como ModBus, CanBus, GPIO, e incluso sistemas de ficheros mediante conectores.

En Greengrass, los disopositivos se comunican de manera segura en una red local e intercambian mensajes entre ellos sin la necesidad de conectarse a la *Cloud.* Proporciona un gestor de mensajes de publicación/subscripción que puede almacenar en un búfer los mensajes si la conectividad se pierde por lo que no se pierden los mensajes entrantes y salientes.

Greengrass protege los datos de usuarios:

- Mediante autenticación y autorización segura de dispositivos.
- Mediante conectivida segura en las redes locales.
- Entre dispositivos locales y *Cloud.*

Las credenciales de seguridad de los dispositivos funcionan en grupo hasta que son revocadas, aunque la conectividad con la *Cloud* se vea interrumpida, por lo que los dispositivos pueden seguir comunicándose localmente de manera segura. Además, proporciona actualizaciones sobre la marcha de funciones Lambda.

Greengrass consiste en:

- Software distribuido
    - AWS Iot Greengrass Core Software
    - AWS Iot Greengrass Core SDK
- Servicios Cloud
    - Aws IoT Greengrass API
- Características
    - Entorno de ejecución Lambda
    - Implementación de *shadows*
    - Gestor de mensajes
    - Gestor de grupos
    - Agente de actualización sobre la marcha
    - Gestor de *stream* de datos

- Acceso a recursos locales
- Interfaz de Machine Learning
- Gestor de sercretos locales
- Conectores con integraciones por defecto de servicios, protocolos y *software*

### 3.3.1 AWS IoT Greengrass Core Software

[20] El software AWS IoT Greengrass Core proporciona las siguientes funcionalidades:

- Implementación y ejecución local de conectores y funciones de Lambda.
- Procesa los flujos de datos de forma local con exportaciones automáticas a AWS Cloud.
- Mensajes MQTT a través de la red local entre dispositivos, conectores y funciones de Lambda mediante suscripciones administradas.
- Mensajes MQTT entre AWS IoT y dispositivos, conectores y funciones de Lambda mediante suscripciones administradas.
- Conexiones seguras entre los dispositivos y la nube mediante la autenticación y la autorización de dispositivos.
- Sincronización de sombras o *shadows* locales de dispositivos. Las sombras o *shadows* se pueden configurar para que se sincronicen con la nube.
- Acceso controlado a los recursos del dispositivo local y el volumen.
- Implementación de modelos de aprendizaje automático entrenados en la nube para la ejecución de la inferencia local.
- Detección automática de direcciones IP que permite a los dispositivos detectar el dispositivo principal de Greengrass.
- Implementación central de la configuración de grupos nuevos o actualizados. Una vez descargados los datos de configuración, el dispositivo principal se reinicia automáticamente.
- Actualizaciones de software (OTA) para funciones de Lambda definidas por el usuario de manera segura y transparente.
- Almacenamiento seguro cifrado de los secretos locales y acceso controlado por conectores y funciones de Lambda.

Las instancias de AWS IoT Greengrass Core se configuran a través de las API de AWS IoT Greengrass que crean y actualizan las definiciones de grupos de AWS IoT Greengrass almacenadas en la nube.

### 3.3.2 Aws IoT Greengrass groups

[20] Un grupo de Greengrass es una colección de configuraciones y componentes, como un Core de Greengrass, dispositivos y suscripciones. Los grupos se utilizan para definir un ámbito de interacción. Por ejemplo, un grupo podría representar la planta de un edificio, un camión o un emplazamiento minero entero. En el siguiente diagrama se muestran los componentes que pueden componer un grupo de Greengrass.

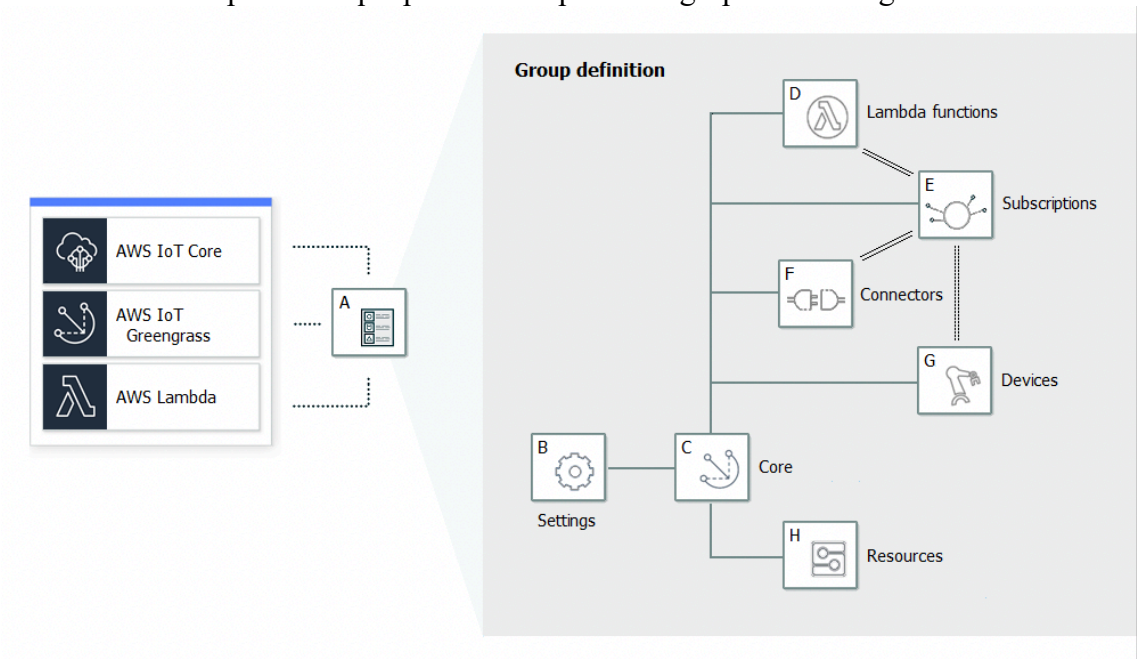



**Figura 16: [20] Greengrass groups**

#### A. Definición de grupo de Greengrass

Información sobre la configuración de grupo y los componentes.

#### B. Configuración de grupo de Greengrass

Entre ellas se incluyen:

- Rol de grupo de Greengrass.
- Autoridad de certificación y configuración de conexión local.
- Información de conectividad del Core de Greengrass.
- Entorno de tiempo de ejecución de Lambda predeterminado. Para obtener más información, consulte Configuración de la creación de contenedores predeterminada para funciones Lambda de un grupo.

- CloudWatch y configuración de registros local. Para obtener más información, consulte Monitorización con registros de AWS IoT Greengrass.

### C. Core de Greengrass

El objeto de AWS IoT (dispositivo) que representa el Core de Greengrass. Para obtener más información, consulte Configuración de AWS IoT Greengrass Core.

### D. Definición de la función de Lambda

Una lista de las funciones de Lambda que se ejecutan localmente en el *Core*, con datos de configuración asociados. Para obtener más información, consulte Ejecutar funciones Lambda en el Core de AWS IoT Greengrass.

### E. Definición de suscripción

Una lista de suscripciones que permiten la comunicación mediante mensajes MQTT.

Una suscripción define:

- Un origen del mensaje y un destino del mensaje. Pueden ser dispositivos, funciones Lambda, conectores, AWS IoT *Core* y el servicio de *shadow* local.

- Un tema o asunto que se utiliza para filtrar mensajes.

### F. Definición de conector

Una lista de los conectores que se ejecutan localmente en el *Core*, con datos de configuración asociados.

Los conectores de Greengrass son módulos precompilados que contribuyen a acelerar el ciclo de vida de desarrollo de escenarios límite comunes. Facilitan la interacción con la infraestructura local, los protocolos de dispositivos, AWS y otros servicios en la nube. El uso de conectores le permitirá dedicar menos tiempo a aprender nuevos protocolos e interfaces API, y más tiempo a centrarse en la lógica importante para su negocio.

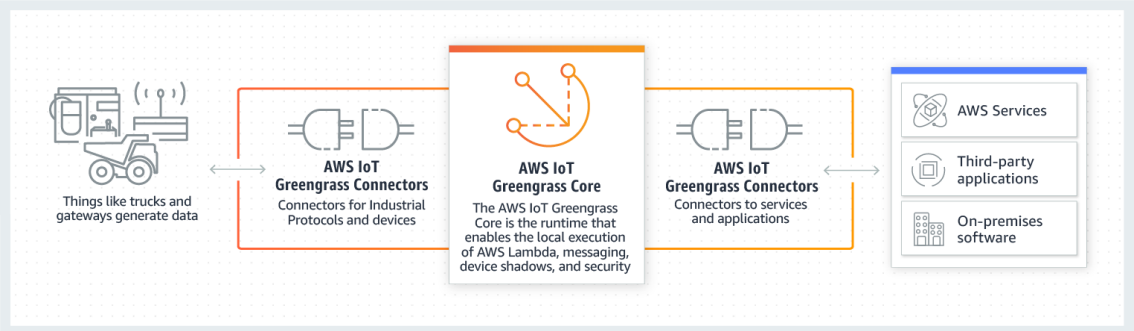



**Figura 17: [20] Conectores en AWS *IoT* Greengrass**

### G. Definición de dispositivos

Una lista de objetos de AWS IoT (dispositivos) que son miembros del grupo de Greengrass, con datos de configuración asociados. Se verá con más detalle en el apartado 3.3.3.

**H. Definición de recursos**

Una lista de los recursos locales, recursos de aprendizaje automático y recursos de secretos en el *Core* de Greengrass, con datos de configuración asociados.

Cuando se implementa, la definición del grupo de Greengrass, las funciones Lambda, los conectores, los recursos y la tabla de suscripción se copian en el dispositivo principal.

### 3.3.3 Dispositivos de AWS *IoT* Greengrass

[20] Un grupo de Greengrass puede contener dos tipos de dispositivos de AWS IoT:

#### 3.3.3.1 Greengrass *Core*

Un *Core* de Greengrass es un dispositivo que ejecuta el software AWS IoT Greengrass *Core*, que le permite comunicarse directamente con AWS *IoT Core* y el servicio AWS IoT Greengrass. Un *Core* tiene su propio certificado de dispositivo que se utiliza para autenticarse con AWS *IoT Core*. Tiene un *shadow* de dispositivo y una entrada en el registro de AWS *IoT Core*.

Los *Cores* de Greengrass ejecutan un tiempo de ejecución de Lambda local, un agente de implementación y un rastreador de direcciones IP que envía información de dirección IP al servicio de AWS IoT Greengrass para permitir a los dispositivos de Greengrass detectar automáticamente su información de conexión de grupo y *Core*.

#### 3.3.3.2 Dispositivo conectado a un *Core* de Greengrass

Los dispositivos conectados (también denominados dispositivos Greengrass) también tienen su propio certificado de dispositivo para la autenticación de AWS IoT *Core*, un *shadow* de dispositivo y una entrada en el registro de AWS IoT *Core*. Los dispositivos Greengrass pueden ejecutar FreeRTOS o utilizar el SDK de dispositivo de AWS IoT o la API de detección de AWS IoT Greengrass para obtener información de detección utilizada para conectarse y autenticarse con el *Core* del mismo grupo de Greengrass.

En un grupo de Greengrass, puede crear suscripciones que permitan a los dispositivos comunicarse a través de MQTT con funciones de Lambda, conectores y otros dispositivos del grupo, y con AWS *IoT Core* o el servicio de sombras local. Los mensajes de MQTT se enrutan a través del *Core*.

Si el dispositivo de *Core* pierde la conectividad con la nube, los dispositivos pueden seguir comunicándose a través de la red local. El tamaño de estos dispositivos puede

variar, desde dispositivos con microcontroladores más pequeños a dispositivos más grandes. En la actualidad, un grupo de Greengrass pueden contener un máximo de 200 dispositivos. Un dispositivo puede ser miembro de hasta 10 grupos.

## 3.4 Escenarios aplicables a este nuevo paradigma

[3] En este apartado se va a repasar el estado de algunas de las aplicaciones *IoT,* pero sobretodo, cuales de ellas se pueden verse más beneficias por este nuevo paradigma de *serverles IoT.*

### 3.4.1 Smart Cities

Las ciudades son ecosistemas complejos, donde la calidad de vida es una preocupación importante. En tales entornos urbanos, las personas, las empresas y las autoridades experimentan necesidades y demandas específicas en ámbitos como la atención médica, los medios de comunicación, la energía y el medio ambiente, la seguridad y los servicios públicos.

Una ciudad se percibe cada vez más como un solo "organismo", que necesita ser monitorizado eficientemente para proporcionar a los ciudadanos información precisa. Las tecnologías de IoT son fundamentales para recopilar datos sobre el estado de la ciudad y difundirlos a los ciudadanos. En este contexto, las ciudades y las áreas urbanas representan una masa crítica a la hora de configurar la demanda de servicios avanzados basados en IoT.

En este punto es donde las implementaciones *serverless* pueden generar un beneficio añadido al actuar como un conjunto de servicios permitiendo procesar los datos con mayor agilidad y establecer actuaciones según la necesidad. La complejidad de una implementación para una solución de *Smart Cities* hace esencial emplear tecnologías que fomenten la facilidad de despliegues, cambios y mantenimiento sin generar una dependencia demasiado fuerte. Por lo tanto, las soluciones *serverless IoT* son una opción beneficiosa para esta aplicación.

### 3.4.2 Industry 4.0

El *IoT* industrial o *IIoT (Industrial IoT)* se refiere al *IoT* empleado en industrias de fabricación, logística, petróleo y gas, transporte, energía, minas y metales, aviación y otras. *IIoT* sigue en una etapa bastante temprana donde aún no se tiene claro de las tecnologías de Internet actualmente y que elementos son aplicables a estas soluciones.

Esto se debe a la complejidad de estas aplicaciones donde respuestas en tiempo real suelen ser críticas para diversas industrias, que actualmente no se cumplen mediante el uso de *Internet.* Esta es una de las razones por las que este tipo de soluciones necesitan de tecnologías de computación *Edge* que permitan reducir los tiempos de respuesta y permitir gestionar un gran número de dispositivos de manera coordinada y estable.

Otro de los problemas y uno de los mayores cuellos de botella es la falta de capacidad de compartir información entre dispositivos debido a que utilicen distintos “lenguajes”. Por lo que, aunque se pueda poner sensores que recojan datos, si no se puede aprovechar para establecer actuaciones en base a estos datos comunicándose con otros

dispositivos no genera un valor añadido más que el mero conocimiento del estado de los procesos. Por lo tanto, es necesaria una estandarización de la comunicación.

A pesar de ello el beneficio potencial del IIoT es enorme. La eficiencia operativa es una de sus principales atracciones, y los primeros esfuerzos se centran en estos beneficios. Al introducir la automatización y técnicas de producción más flexibles, por ejemplo, los fabricantes podrían aumentar su productividad hasta en un 30%. En este contexto, se deben dominar tres capacidades de IIoT:

- Computación controlada por sensores, convirtiendo los datos recibidos por los sensores en información que permita mejorar la percepción de los procesos
- Analítica industrial, transformando los datos recibidos de los sensores en acciones perceptibles
- Aplicaciones de máquinas inteligentes, integrando los sensores y componentes inteligentes en las máquinas.

Viendo todos los puntos analizados para *IIoT* se puede concluir que las ventajas introducidas por la tecnología *serverless IoT* son claras. Permitiendo reducir los tiempos de respuesta al tener los nodos de computación en el *edge,* se tiende a una estandarización con dispositivos como servicio y permite establecer patrones de analítica de datos en tiempo real para realizar actuaciones en base a los datos recibidos. Todo ello sin la necesidad de estar completamente conectados a internet, mientras puedan establecer una comunicación directa entre los dispositivos y el nodo *Edge.*

### 3.4.3 *Smart Home*

Según se ha ido viendo, hay una mayor inversión en el área de hogares inteligentes o *Smart Home,* han ido apareciendo aplicaciones de automatización del hogar. Aún así, siempre han ido enfocadas a públicos específicos que ha provocado la creación de segmentos verticales desconectados.

Como ya se ha mencionado anteriormente, dentro del concepto *IoT* uno de los factores más importantes es evitar segmentos que impidan tener dispositivos compatibles independientemente de la solución. Por lo tanto, en este ámbito la posibilidad de establecer una red de dispositivos como servicio que interactúen entre sí independientemente del propósito para soluciones *Smart Home.*

Entre las soluciones que han ido surgiendo aparecen las que están relacionadas con la seguridad del hogar, eficiencia energética y el ahorro de engería. Estás se en impulsadas por innovaciones en la luz y control de la sala, *IoT* fomentará el desarrollo de aplicaciones infinitas para la automatización del hogar.

En general, las soluciones de automatización de edificios están comenzando a converger y también se están moviendo, desde las aplicaciones actuales de lujo, seguridad y confort, a una gama más amplia de aplicaciones y soluciones conectadas; Esto creará oportunidades de mercado. Si bien las soluciones para el hogar inteligente de hoy están fragmentadas, se espera que el *IoT* conduzca a un nuevo nivel de interoperabilidad entre el hogar comercial y las soluciones de automatización de edificios.

Justo aquí es donde el concepto de *serverless IoT* puede ser determinante ya que permitirá introducir un aislamiento a nivel de dispositivos empleados, mayor facilidad de desarrollo y despliegue de aplicaciones, sin olvidar de las ventajas que puede aportar en un entorno como el hogar. En gran medida no es necesario tener una comunicación directa con los servicios *Cloud,* por lo tanto, se puede establecer una red de dispositivos dinámica que permita mediante la computación *edge* una interacción rápida y eficiente que proporcione una buena experiencia de usuario.

Además, se puede beneficiar de complementarse con servicios *Cloud* para establecer patrones de uso que permitan ajustar determinadas rutinas de automatización para permitir ahorrar o generar una comodidad añadida en los hogares. Otro factor importante es que debido a las cualidades de la computación *edge* del *serverless IoT* se pueden desplegar sistemas complejos sin realizar inversiones demasiado grandes.

Por ello, se va a utilizar esta aplicación *IoT* como factor de estudio para la implementación de una aplicación *serverless IoT.* Analizando la complejidad y los beneficios que puede aportar para aplicarlos a este caso de uso.

## 3.5 Cuál es el estado del arte IoT

En base a los estudios realizados por alumnos de la UOC en el ámbito *IoT* de soluciones más clásicas voy a establecer un breve estado del arte de las tecnologías actualmente más utilizadas. Esto permitirá compararlo con está nueva tendencia tecnológica de *serverless IoT*.

### 3.5.1 Gateway ***IoT***

Para empezar, es interesante exponer los conceptos expuestos en [21] del diseño conceptual de un *gateway* diseñado para poder tenerlo como referencia para compararlo con el Greengrass *Core*. En este proyecto se refleja claramente una arquitectura *IoT* con todos los posibles componentes de comunicación *IoT.* Lo voy a usar como base sin animo de lucro para la comparación con la arquitectura de AWS Greengrass *IoT*

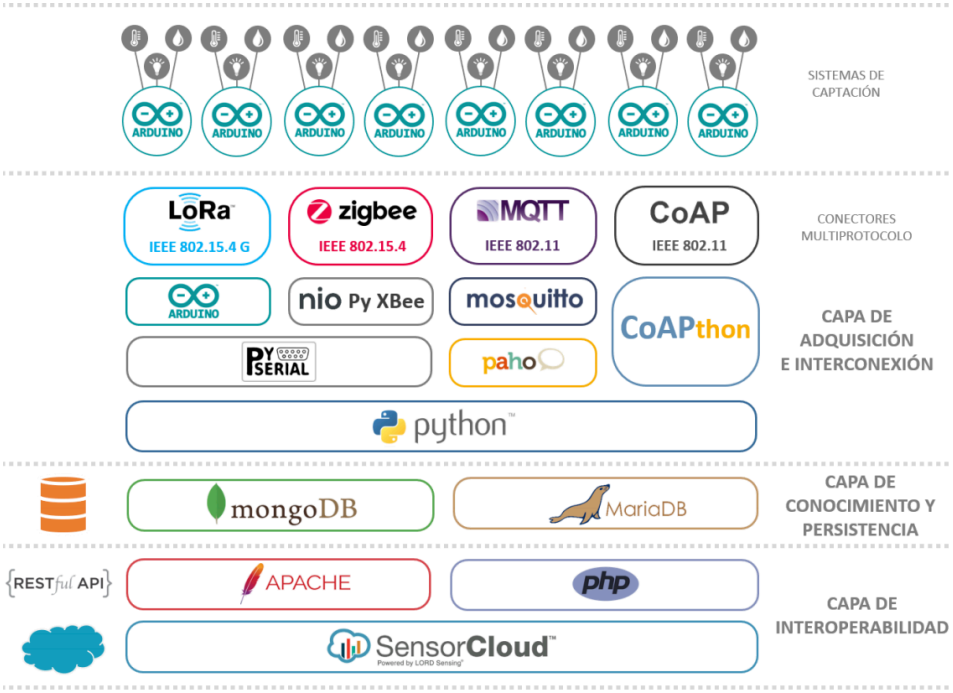
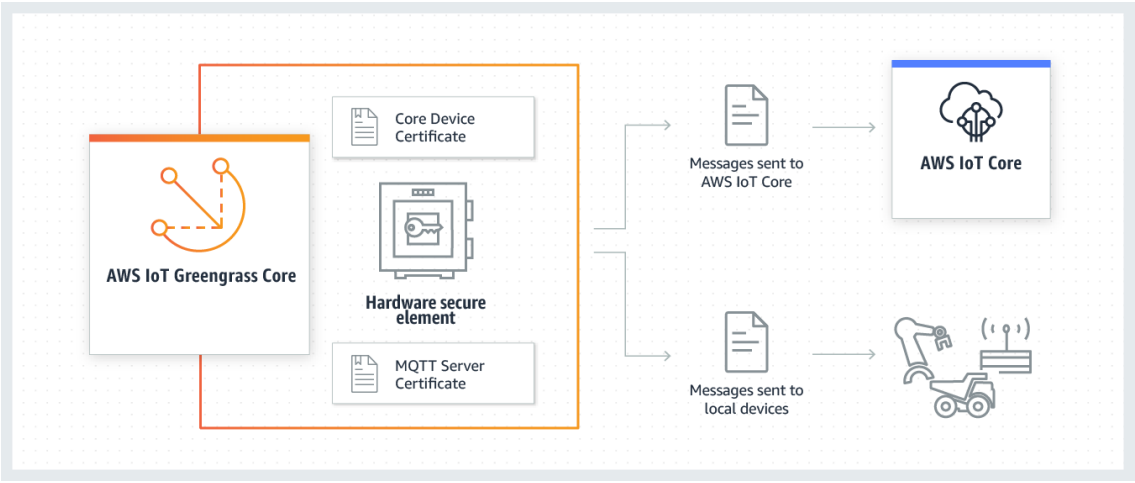



**Figura 18: [21] Arquitectura en capas de la solución a comparar**

Aquí se puede ver la alta complejidad de todas las capas introducidas para poder establecer un *Gateway IoT* y todos los componentes necesarios. Mediante la utilización de AWS *IoT* Greengrass se permite un nivel de abstracción del Gateway que permite facilitar su despliegue e implementación. Esto conlleva a que pueda ser desplegado en cualquier dispositivo independientemente del sistema operativo, esto último es parcialmente cierto ya que debe ser un sistema operativo Linux, pero aún así, es posible contenrizarlo para desplegarlo como un contenedor Docker. Esto da la posibilidad de desplegarlo en cualquier dispositivo (incluido Windows y MacOS si se utiliza docker desktop) que tenga un *daemon* de Docker.




### 3.5.2 Comparativa de Plataformas *Software*

[22] En otra memoria empleada para el estudio de plataformas *IoT* se establece una comparativa de las distintas plataformas *software*. En base a lo que se expone en ese

proyecto voy a añadir, sin animo de lucro, la Plataforma AWS *IoT* Greengrass para poder comprarar las distintas plataformas que había con la que proporciona está nueva solución *serverless IoT.*

| Plataforma | *Hardware* | Ámbito | Ventajas | Desventajas |
|---|---|---|---|---|
| **Thingspeak** | Arduino, Spark, Raspberry Pi, Electronic Imp | Smart Home Prototipos | Interfaz. App. Integración redes sociales | Documentación limitada a cierto HW. |
| **Carriots** | Arduino Raspberry Electronic Imp Beagle TST Industrial | City Energy Oil Agiculture Buildings Retail Banking Consumer Loguistic | Integración redes sociales Hw compatible Ámbitos de aplicación. | Poca Documentación |
| **Spark** | Spark | Smart Home Prototipos | Ideal para iniciarse. Escalable Comunidad al alza. | Compatibilidad HW |
| **Electronic Imp** | Electronic Imp. Otros | Loguistic Consumer Home Industry | HW. Ámbitos de aplicación Ecosistema propio. Escalable. | No recomendable para iniciarse. |
| **Bluelabs** | SATEL Grid | City Energy Manufacturing | Especialistas en Smart City. Aplicaciones orientadas a Smart City. | HW limitado |
| **Bluelabs** | SATEL Grid | City Energy Manufacturing | Especialistas en Smart City. Aplicaciones orientadas a Smart City. | HW limitado. |
| **Thinking Things** | Propietario | Smart Home Prototipos | Fácil Configuración. Integración fácil entre HWplataforma Ideal para prototipos. Ideal para iniciarse | Poca variedad Sensores. Sol es posible usar su HW. |
| **AWS IoT Greengrass** | Linux y contenedores | Cualquiera | Abstracción del HW, Sencillo, Lambda, Servicios Cloud, Edge | Depende de una licencia AWS |

## 3.6 Problema a resolver

Como se ha expuesto en el apartado 3.4.3 las soluciones de *Smart Home* están recibiendo actualmente un interés especial, por lo tanto, estudiar como la tecnología *serverless IoT* puede beneficiar a estas aplicaciones es un factor interesante.

Está solución se puede beneficiar de complementarse con servicios *Cloud* para establecer patrones de uso que permitan ajustar determinadas rutinas de automatización para permitir ahorrar o generar una comodidad añadida en los hogares. Otro factor importante es que debido a las cualidades de la computación *edge* del *serverless IoT* se pueden desplegar sistemas complejos sin realizar inversiones demasiado grandes.

Se va a estudiar la posibilidad de implementar un caso de uso real simulado donde se pueda establecer la complejidad de gestión, configuración y despliegue. Además, poder poner a prueba el concepto de *serverless IoT.* Consiguiendo establecer un conjunto de dispositivos que se comunican mediante MQTT y que se benefician de servicios proporcionados por la *Cloud,* por ejemplo, bases de datos relacionales distribuidas como DynamoDB.

Además, con los datos agregados que se han recolectado se puede establecer un modelo de datos a entrenar que permitirá cargarlo en el dispositivo y así poder mejorar los procesos en base a los datos analizados.

La idea será medir distintos factores clave para una casa, programar temporizadores y distintas funcionalidades. Como prueba de valor se ha empleado un sistema para medir la temperatura ambiente para programar calefacción/aire acondicionado a la temperatura ideal ahorrando lo máximo.

# 4 Planteamiento de la solución

## 4.1 Qué se quiere resolver

Se quiere resolver un caso de uso real, en este caso una aplicación *Smart Home,* que permita establecer el valor que puede aportar una arquitectura *serverless IoT* y validar está solución.

Hay diversas soluciones aplicadas a *Smart Home* que pueden ser interesantes abordar. Como primeras ideas surge buscar una manera de establecer lógicas en elementos comunes que puedan favorecer el ahorro del consumo y, a la vez, generar comodidades para casos cotidianos que mejoran la percepción de estos elementos *IoT*.

En concreto, la idea es desarrollar una simulación de un regulador de temperatura que mediante una serie de sensores pueda estar constantemente recopilar información de la temperatura ambiente. Con ello, sería posible poder establecer patrones de actuación para regular la temperatura con el apoyo de un calefactor y un climatizador. Ajustando el comportamiento de ambos dispositivos en función de las necesidades, siempre intentando reducir el consumo y asegurar una temperatura estable.

El factor más relevante es el poder estar continuamente recopilando datos del entorno, que permitan establecer patrones de actuación. Estos elementos han de estar continuamente intercambiando información de manera *offline* para que no se vea afectada la estabilidad de la solución si hay un corte de Internet. A la vez, se intenta establecer una lógica de funcionalidad en base a límites y parámetros establecidos

El punto más diferenciador es la posibilidad de estar trabajando continuamente con elementos *Cloud* como un apoyo, no una dependencia. Es decir, poder estar continuamente subiendo datos en una base de datos en la *Cloud* que no afecte en tiempo real la implementación, pero, mediante elementos de analítica e inteligencia artificial se puedan establecer patrones que permitan mejorar la solución.

Por lo tanto, aplicando modelos de *Machine Learning* utilizando servicios que proporciona la *Cloud*. Pudiendo establecer un sistema de aprendizaje automático que permita establecer modificaciones de la implementación para ajustarse a patrones de utilización, mejorar la eficiencia y, en definitiva, mejorar la solución.

## 4.2 Qué implementación puede emular el escenario

Según lo que se ha ido viendo en apartados anteriores se puede establecer un escenario que pueda implementar la solución a resolver expuesta en el apartado anterior. Inicialmente será necesario establecer un nodo de computación *Edge* que permita realizar toda la lógica y control de los distintos dispositivos que interactúan entre sí. Estableciendo como concepto fundamental la arquitectura *serverles IoT,* generando, en la manera de lo posible, dispositivos como servicio que permitan estandarizar las interacciones.

Está claro que los componentes más importantes son los dispositivos y el nodo de computación *Edge*, pero hay que tener en cuenta todos los elementos que permiten la interacción entre ellos. En el siguiente apartado se enumerarán todos los elementos clave que permitan establecer esta solución. Entre los que aparecen las subscripciones, encargadas de generar las interacciones entre dispositivos mediante un modelo publicación/suscripción con MQTT.

Por otro lado, será necesario poder desplegar funciones Lambda que permitan generar toda la lógica *serverless* en el nodo de computación *Edge*. Este nodo, por las características de este tipo de arquitectura es independiente y puede ser tanto un ordenador, como una Raspberry PI e incluso una máquina virtual o un contenedor.

Los dispositivos necesitarán interactuar con los datos en tiempo real de cada uno de los dispositivos, por lo tanto, aparece el concepto de *Shadow devices*. [18] La sombra de un dispositivo es un documento JSON que se utiliza para almacenar y recuperar información del estado actual de un dispositivo mediante MQTT o HTTP.

En base a todos estos elementos clave se puede emular el caso expuesto en el apartado anterior, 4.1. Por lo tanto, en el siguiente apartado se expondrán los distintos elementos clave y su configuración para poder establecer este planteamiento.

## 4.3 Elementos clave empleados

### 4.3.1 AWS *IoT* Greengrass *Group*

Como se ha visto en el apartado 3.3.2, un grupo de Greengrass es una colección de configuraciones y componentes, como un *Core* de Greengrass, dispositivos y suscripciones. Los grupos se utilizan para definir un ámbito de interacción. Por ejemplo, un grupo podría representar la planta de un edificio, un camión o un emplazamiento minero entero. En el siguiente diagrama se muestran los componentes que pueden componer un grupo de Greengrass.

Esto se hace a través del servicio proporcionado por la plataforma *Cloud* de AWS. Esto genera un servicio que permite configurar:

- Subscripciones
- *Core*
- Dispositivos
- Lambdas
- Recursos
- Conectores

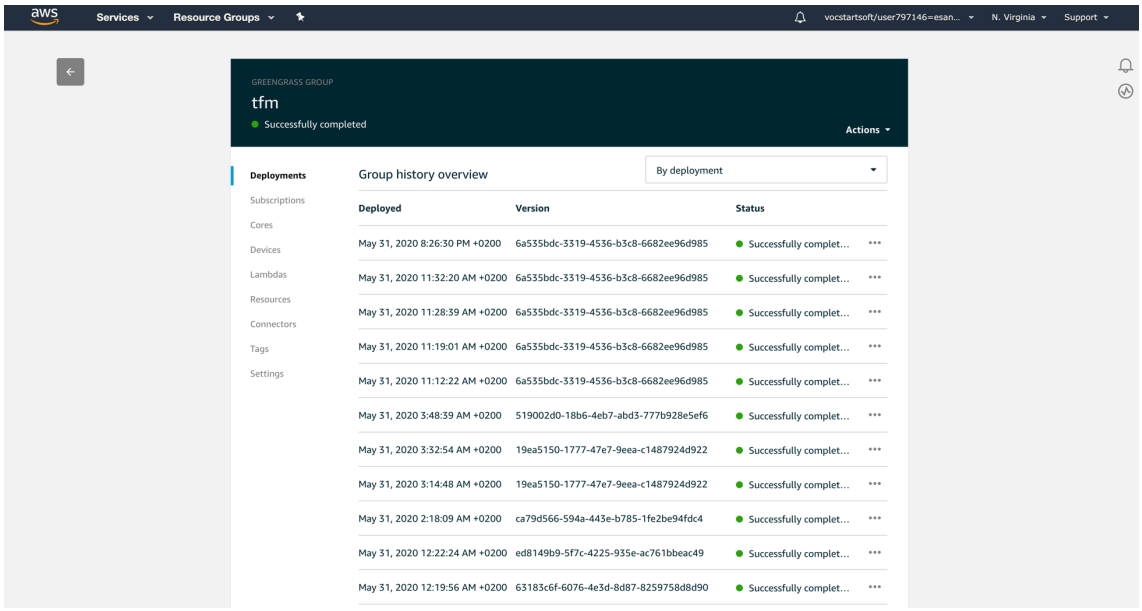



**Figura 19: AWS *IoT* Greengrass *Group***

## 4.3.2 Dispositivos

### *4.3.2.1 Core*

Este dispositivo contiene el *software* AWS *IoT* Greengrass Core que proporciona todas las funciones necesarias para poder establecer la lógica de la solución. Permite la ejecución de funciones Lambda, MQTT, dispositivos *shadow* que interactuará directamente con la *Cloud* y los distintos dispositivos. Este se ejecuta localmente y permite conectividad offline.

Este *software* permite establecer como dispositivo AWS *IoT* Greengrass prácticamente cualquiera que utilice Linux o, en su defecto, un contenedor. Siguiendo los pasos de instalación del manual [18] se puede poner en marcha un dispositivo *Core* en menos de media hora.

Como dispositivo *Core* se ha utilizado una Raspberry Pi por las ventajas que aporta en cuanto a tamaño, computación y desplegabilidad. Como la idea principal de este tipo de arquitectura es poder abstraerse de la infraestructura subyacente no se va a exponer las características de una Raspberry Pi, ya que debería ser completamente transparente como parte del concepto *serverless IoT*.

### *4.3.2.2 Things*

Estos dispositivos son los encargados de establecer la comunicación con el medio físico. Dándolos de alta en el grupo del servicio de Greengrass de AWS se genera un certificado para poder establecer la comunicación con este dispositivo. Como se ha visto en el apartado 3.3.3.2 estos dispositivos pueden ejecutar FreeRTOS o utilizar el SDK de dispositivo de AWS IoT o la API de detección de AWS IoT Greengrass para obtener información de detección utilizada para conectarse y autenticarse con el *Core* del mismo grupo de Greengrass.

Para la esta implementación de Smart Home es necesario generar los siguientes dispositivos:

- Regulador y sensor de temperatura (Temperature_Sensor)
- Climatizador (cooler)
- Calefactor (heater)

Es pueden ver los distintos dispositivos generados:

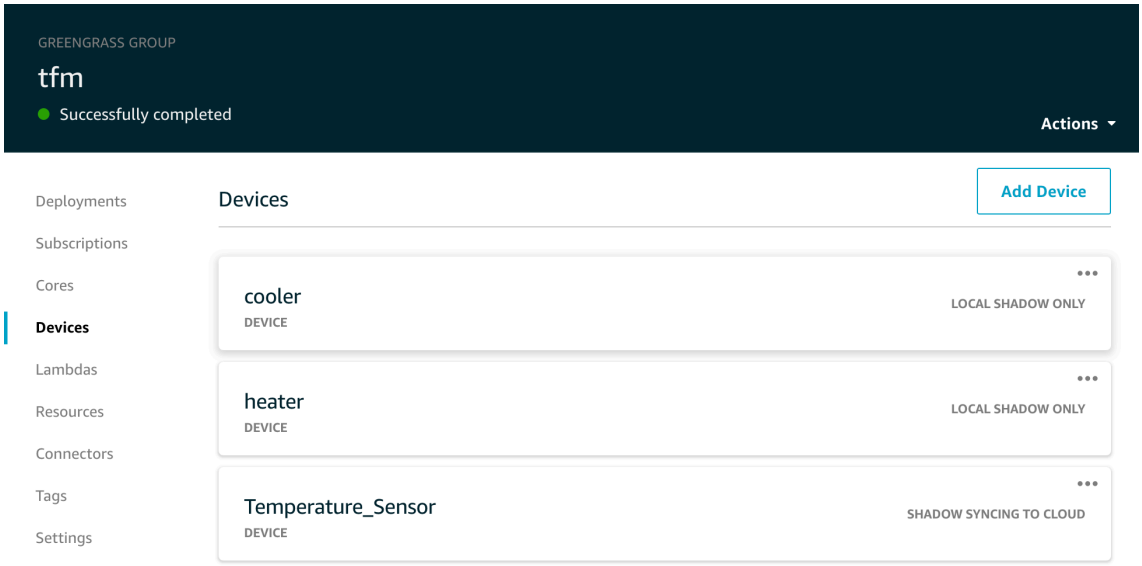



**Figura 20: Dipositivos Greengrass**

### 4.3.3 Shadows

[18] Este dispositivo es un documento JSON que se utiliza para almacenar y recuperar información del estado actual de un dispositivo. El servicio Device Shadow mantiene una sombra para cada dispositivo que se conecta a AWS IoT. La sombra se puede utilizar para obtener y establecer el estado de un dispositivo a través de MQTT o HTTP, sin tener en cuenta si el dispositivo está conectado o no a Internet. La sombra de cada dispositivo se identifica de forma inequívoca mediante el nombre del objeto correspondiente.

Para este caso se ha empleado una única sombra para el regulador de temperatura que permitirá conocer la temperatura actual, la temperatura de referencia y si el calefactor o climatizador están activos. Contiene dos parámetros, el estado deseado y el estado reportado. Esto implica que primero se actualiza el estado deseado y luego se reporta si se ha hecho efecto. Con lo que se puede, en todo momento, saber el estado actual y el nuevo estado del *shadow* del dispositivo.

**Shadow state:**

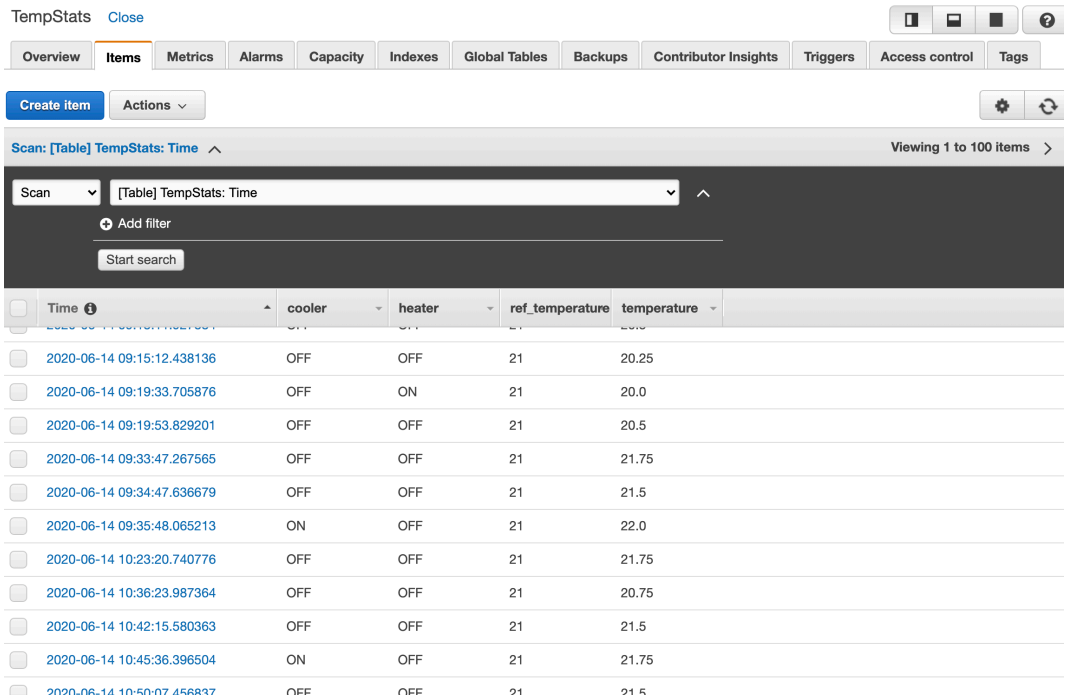

```
{
  "desired": {
    "ref_temperature": "21",
    "heater": "OFF",
    "temperature": "21.25",
    "cooler": "OFF"
  },
  "reported": {
    "temperature": "21.25",
    "ref_temperature": "21",
    "heater": "OFF",
    "cooler": "OFF"
  }
}
```


**Figura 21: *Shadow* del regulador de temperatura o sensor**

### 4.3.4 AWS Services

Es posible interactuar con servicios *Cloud* directamente a través del *Core.* Se pueden configurar para que una función Lambda local interactúe con cualquier elemento dentro del Greengrass *Group*. Mediante una subscripción es posible establecer una comunicación con MQTT que permita enviar datos o interactuar con servicios *Cloud* y los dispositivos.

Se ha utilizado una función Lambda que permite generar una tabla en una base de datos no relacional DynamoDB donde se va almacenando el historial de cambios de temperatura, temperatura de referencia y el estado de los dispositivos. Con lo que poder explotar estos datos para aplicar modelos de Machine Learning que permitan un aprendizaje continuo de la evolución del a temperatura.




| Time | cooler | heater | ref_temperature | temperature |
|---|---|---|---|---|
| 2020-06-14 09:15:12.438136 | OFF | OFF | 21 | 20.25 |
| 2020-06-14 09:19:33.705876 | OFF | ON | 21 | 20.0 |
| 2020-06-14 09:19:53.829201 | OFF | OFF | 21 | 20.5 |
| 2020-06-14 09:33:47.267565 | OFF | OFF | 21 | 21.75 |
| 2020-06-14 09:34:47.636679 | OFF | OFF | 21 | 21.5 |
| 2020-06-14 09:35:48.065213 | ON | OFF | 21 | 22.0 |
| 2020-06-14 10:23:20.740776 | OFF | OFF | 21 | 21.75 |
| 2020-06-14 10:36:23.987364 | OFF | OFF | 21 | 20.75 |
| 2020-06-14 10:42:15.580363 | OFF | OFF | 21 | 21.5 |
| 2020-06-14 10:45:36.396504 | ON | OFF | 21 | 21.75 |

**Figura 22: Tabla de historial de datos de temperatura**

### 4.3.5 Resources

#### 4.3.5.1 GPIO

Es posible acceder a recursos locales del dispositivo, en este caso solo se ha utilizado para poder trabajar con los pines GPIO de la Raspeberry PI. En concreto, es necesario especificar la ruta de acceso al dispositivo "/dev/gpiomem" que permite a las funciones Lambda trabajar con esos recursos bajo eventos de lectura o escritura.

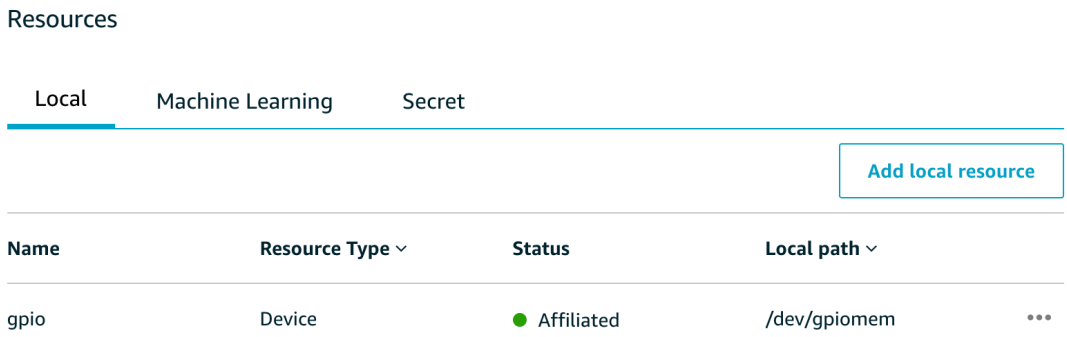



**Figura 23: Recursos locales, GPIO**

Para el caso expuesto se ha utilizado dos pins para activar/desactivar el calefactor o el climatizador. En concreto el pin 26 para el calefactor y el pin 21 para el climatizador.

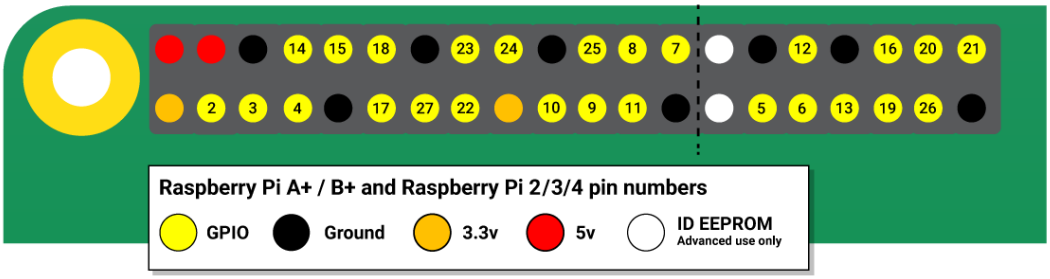



**Figura 24: [23] Raspberry Pi Pins**

Para simular esos dos dispositivos, se han utilizado dos leds como forma de comprobar visualmente cuando se ha encendido uno de los dispositivos. El led azul representa el climatizador y el pin rojo representa el calefactor. Se entiende que, en vez de leds deberían estar conectados en esos pines los activadores de los dispositivos físicos de calefacción y climatización.

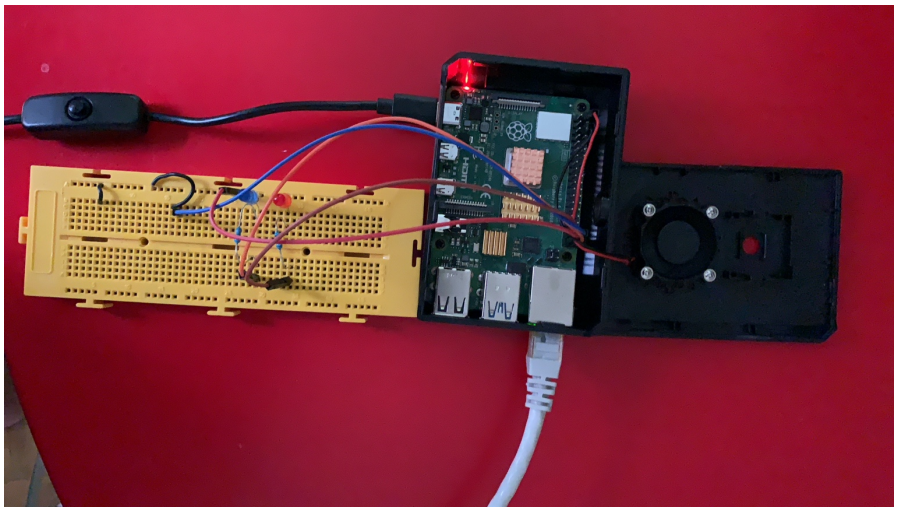


**Figura 25: Conexión de pins de Raspberry Pi con leds**

### 4.3.5.2 Machine Learning

[18] AWS IoT Greengrass le permite realizar la inferencia de aprendizaje automático (ML) en el *edge* con datos generados localmente utilizando modelos entrenados en la *Cloud*. Permite beneficiarse de la baja latencia y el ahorro de costes que supone la ejecución de locales, aprovechando al mismo tiempo la potencia de computación de la *Cloud* para el entrenamiento de modelos y el procesamiento complejo.

Es posible entrenar los modelos de datos locales en cualquier lugar, implementarlos localmente como recursos de aprendizaje automático en un grupo de Greengrass y, a continuación, obtener acceso a ellos desde funciones de Lambda de Greengrass. Por ejemplo, puede crear y entrenar modelos de aprendizaje profundo en Amazon SageMaker e implementarlos en el núcleo de Greengrass. A continuación, las funciones de Lambda pueden utilizar los modelos locales para realizar inferencias con los dispositivos conectados y enviar datos de entrenamiento nuevos de vuelta a la nube.

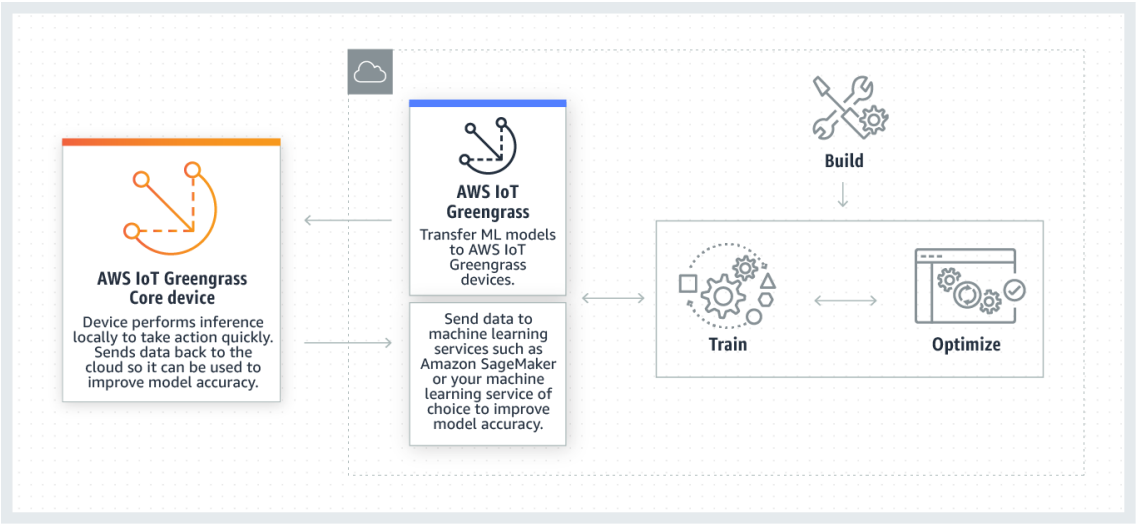



**Figura 26: Working flow of Machine Learning**

Lamentablemente, no ha sido posible aplicar modelos de Machine Learning integrados con la solución expuesta, pero al ser un factor muy relevante y diferenciador es importante al menos mencionarlo. La idea es poder utilizar los datos trasladados a la

*Cloud* que permita optimizar la activación de los dispositivos de calefacción y climatización con lo que, se consigue ahorrar costes innecesarios y estimar los tiempos de activación óptimos. Además, es posible entender patrones de uso que permita ajustar o preparar los dispositivos analizando los datos.

### 4.3.6 Lambda

Estas son funciones como servicio ejecutadas en los dispositivos que permiten establecer la lógica funcional mediante eventos MQTT del sistema. Es posible establecer los límites de memoria y *timeouts* de las funciones Lambda, además de establecer variables de entorno que necesiten para la ejecución del código.

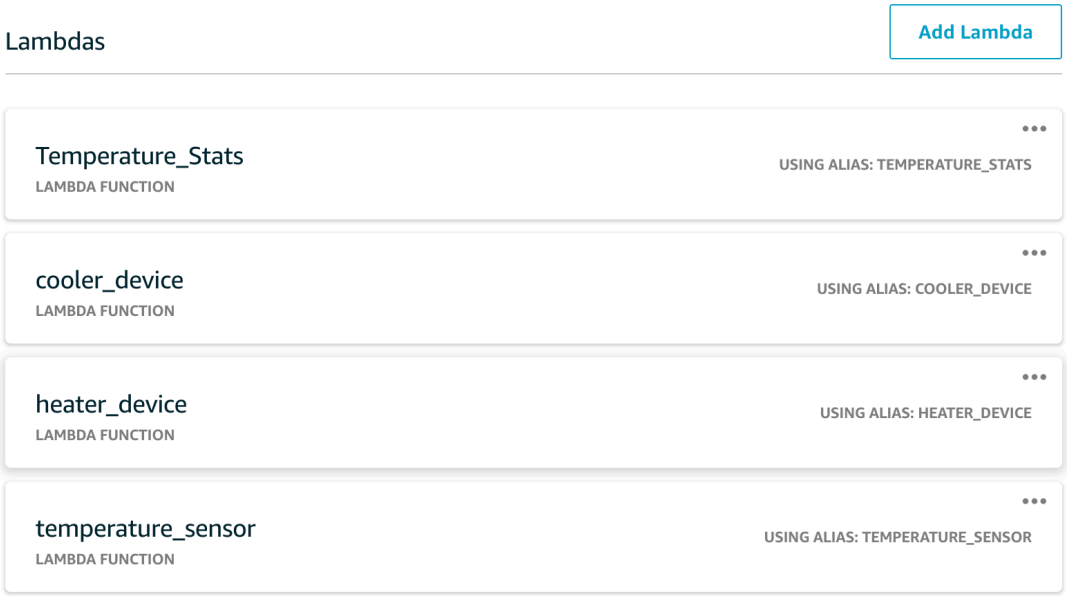



**Figura 27: Funciones Lambda**

Todo el código se puede ver en el anexo, en este apartado se expondrán partes claves para entender la lógica de las funciones.

#### 4.3.6.1 Temperature_Stats

Esta función se encarga de subir los datos actualizados en un DynamoDB, cada vez. que recibe un evento.

```
def function_handler(event, context):

    logger.info(event)
    data = event["current"]["state"]["reported"]
    if data:
        global tableName
        logger.info(f'Cars passed during green light:
{json.dumps(event["current"]["state"]["reported"])}' )
        table = dynamodb.Table(tableName)
        data_prot = {
                "Time": str(datetime.utcnow())
                }
        data_prot.update(data)
```

```
        table.put_item(
            Item=data_prot
        )
    return
```

#### 4.3.6.2 Cooler_device

Este se encarga de comprobar el estado actual del pin asociado al climatizador, abstrayéndose completamente del pin configurado. Simplemente recibe una actualización del estado actual en base a la suscripción configurada. En caso de estar activo actualiza el *shadow* para reflejarlo.

```
def function_handler(event, context):
    if event:
        cooler = "ON"
    else:
        cooler = "OFF"
    JSONPayload = '{"state":{"desired":{"cooler":' + '"' + str(cooler) +
'"}}}'
    print(JSONPayload)
    deviceShadowHandler.shadowUpdate(JSONPayload,
customShadowCallback_Update_Desired, 5)
    return
```

#### 4.3.6.3 Heater_device

Este se encarga de comprobar el estado actual del pin asociado al calefactor, abstrayéndose completamente del pin configurado. Simplemente recibe una actualización del estado actual en base a la suscripción configurada. En caso de estar activo actualiza el *shadow* para reflejarlo.

```
def function_handler(event, context):
    if event:
        cooler = "ON"
    else:
        cooler = "OFF"
    JSONPayload = '{"state":{"desired":{"cooler":' + '"' + str(cooler) +
'"}}}'
    print(JSONPayload)
    deviceShadowHandler.shadowUpdate(JSONPayload,
customShadowCallback_Update_Desired, 5)
    return
```

#### 4.3.6.4 Temperature_Sensor

El código es más complejo y extenso, por lo que, lo mejor es acudir al Anexo para verlo detalladamente. Lo principal es la lógica que hay detrás para simular un sensor de

temperatura. En base a una temperatura de referencia se establece una distribución de pesos de probabilidad para actualizar la temperatura actual. Si la temperatura de referencia está muy por encima (± 2ºC) la probabilidad de que varíe hacia la temperatura deseada aumenta. Cuando llega a una temperatura menor/mayor 2ºC de la temperatura de referencia la distribución de pesos se vuelve menos brusca, pero tendiendo a favorecer la variación de temperatura hacía la referencia.

Una vez se llega a un a una temperatura ±1ºC se desactiva el calefactor/climatizador y se hace una distribución de pesos que favorece continuar hasta la temperatura de referencia. Si está temperatura llega a ±0,25ºC se entiende que la probabilidad de que la temperatura se mantenga fija incrementa, manteniendo una mínima posibilidad de que se salga de lo establecido para favorecer la simulación de un caso real.

```
def heater_cooler_checker():
    global temperature
    global ref_temperature
    global weights
    global heater
    global cooler
    thingName = "tfm_Core"
    thingName_cooler = "cooler"
    thingName_heater = "heater"
    cooler_gpio = 21
    heater_gpio = 26

    print(f"Weights:{weights}")
    print(f"Cooler: {str(cooler)}")
    print(f"Heater: {str(heater)}")

    if temperature >= ref_temperature + 2:
        print("Cooler ON")
        if not cooler:
            set_gpio_state(cooler_gpio, 1, thingName)

        if heater:
            set_gpio_state(heater_gpio, 0, thingName)
        weights = [50,30,10,10,0]

    elif temperature <= ref_temperature - 2:
        print("Heater ON")

        if not heater:
            set_gpio_state(heater_gpio, 1, thingName)


        if cooler:
            set_gpio_state(cooler_gpio, 0, thingName)
        weights = [0,10,10,30,50]

    elif temperature >= ref_temperature + 1:
        print("Cooler ON 1/2")
```

```
        if not cooler:
            set_gpio_state(cooler_gpio, 1, thingName)

        if heater:
            set_gpio_state(heater_gpio, 0, thingName)
        weights = [30,50,10,10,0]

    elif temperature <= ref_temperature - 1:
        print("Heater ON 1/2")

        if not heater:
            set_gpio_state(heater_gpio, 1, thingName)


        if cooler:
            set_gpio_state(cooler_gpio, 0, thingName)
        weights = [0,10,10,50,30]

    elif temperature >= ref_temperature - 0.25 and temperature <=
ref_temperature + 0.25:
        print("Reference temperature reached")
        weights = [10,20,40,20,10]

    else:
        if cooler:
            print("Cooler OFF")
            set_gpio_state(cooler_gpio, 0, thingName)
            weights = [10,30,40,20,0]

        if heater:
            print("Heater OFF")
            set_gpio_state(heater_gpio, 0, thingName)
            weights = [0,20,40,30,10]

    read_gpio_state(cooler_gpio, thingName_cooler)
    read_gpio_state(heater_gpio, thingName_heater)
```

El código que establece las actualizaciones viene dado por:

```
def temperature_checker():
    global temperature
    global choices
    global weights
    """
    Simulates a read of temperature sensor GPIOs
    """
    temperature += random.choices(choices, weights=weights, k=1)[0]
    heater_cooler_checker()
    JSONPayload = '{"state":{"desired":{"temperature":' + '"' +
str(temperature) + '"}}}'
    print(JSONPayload)
```

```
    deviceShadowHandler.shadowUpdate(JSONPayload,
customShadowCallback_Update_Desired, 5)
    deviceShadowHandler.shadowGet(customShadowCallback_Get, 5)
    Timer(10, temperature_checker).start()
```

### 4.3.7 Subscripciones

En un grupo de Greengrass, se pueden crear suscripciones que permitan a los dispositivos comunicarse a través de MQTT con funciones de Lambda, conectores y otros dispositivos del grupo, y con AWS IoT *Core* o el servicio de sombras local. Los mensajes de MQTT se enrutan a través del *Core*.

Estás subscripciones están estandarizadas para recursos y dispositivos conocidos, pero se puede establecer un patrón de subscripción personalizado. Para el caso utilizado solo ha sido empleado un *topic* de suscripción personalizado que permita cambiar la temperatura de referencia.

[18] AWS IoT Greengrass utiliza una tabla de suscripciones para definir cómo se intercambiarán los mensajes de MQTT. Cada suscripción especifica un origen, un destino y un tema de MQTT (o asunto) a través del cual se envían o reciben los mensajes. AWS IoT Greengrass únicamente permite enviar mensajes de un origen a un destino si se ha definido la suscripción correspondiente.

Una suscripción define el flujo de mensajes en una sola dirección, del origen al destino. Para admitir el intercambio de mensajes bidireccional, debe crear dos suscripciones, una para cada dirección.

En el anexo 9.1 se puede ver la lista completa de subscripciones que debido a su extensión se hace referencia. Si se quiere ver la lista completa se puede encontrar en ese apartado. En el siguiente apartado expondré todas las subscripciones e interacciones.

## 4.4 Interacciones

Una vez se ha conseguido establecer los elementos clave que van a componer la solución, hay que definir las interacciones entre los distintos dispositivos. En este apartado se explorarán todas las interacciones entre los dispositivos en base a las subscripciones definidas. Lo primero es saber que se necesita de cada uno de los dispositivos y luego definir las subscripciones que representarán las interacciones mediante MQTT.

Para explicarlo se van a ir analizando cada uno de los elementos y que prototipo de *topics* utilizan. Como ya se ha dicho la lista completa se puede encontrar en el anexo (apartado 9.1), el primer componente es el origen, el segundo el destino y por último el *topic* al que están subscritos.

### 4.4.1 *Shadows y dispositivos*

Para interactuar con los elementos *shadows* es necesario que el dispositivo publique un mensaje en determinados *topics*. En el caso de querer actualizar un parámetro deberá

publicar un mensaje /update, en caso de que se acepte esa actualización el dispostivo recibirá un /accepted y en caso contrario un /rejected.

| Origen | Destino | *Topic* |
|---|---|---|
| **Dispositivo** | *Shadow* | $aws/things/thing-name/shadow/update |
| ***Shadow*** | Dispositivo | $aws/things/thing-name/shadow/update/rejected |
| ***Shadow*** | Dispositivo | $aws/things/thing-name/shadow/update/accepted |
| ***Shadow*** | Dispositivo | $aws/things/thing-name/shadow/update/delta |
| **Dispositivo** | *Shadow* | $aws/things/thing-name/shadow/get |
| ***Shadow*** | Dispositivo | $aws/things/thing-name/shadow/get/rejected |
| ***Shadow*** | Dispositivo | $aws/things/thing-name/shadow/get/accepted |

### 4.4.2 *Shadows y Lambda*

En este caso se publican mensajes de actualización de los *Shadows* para que sean procesados por una función Lambda, que insertará esos datos en una base de datos DynamoDB.

| Origen | Destino | *Topic* |
|---|---|---|
| ***Shadow*** | Lambda | $aws/things/thing-name/shadow/update/documents |

### 4.4.3 *Lambda y Connector* GPIO

Esto permite a la función Lambda interacturar con los pins, para la lectura del pin es necesario publica un mensaje /read primero y esperar un /state. Para cambiar el estado el pin hay que publicar un mensaje en /write con el valor a cambiar.

| Origen | Destino | *Topic* |
|---|---|---|
| **Lambda** | GPIO | gpio/core-name/pin-number/read |
| **GPIO** | Lambda | gpio/core-name/pin-number/state |
| **Lambda** | GPIO | gpio/core-name/pin-number/write |

### 4.4.4 *IoT Cloud y Lambda*

Este *topic* es personalizado y se utiliza para testear los cambios de la temperatura de referencia, con una función Lambda lo actualiza según el valor recibido. Es necesario que contenga el siguiente mensaje, siendo necesario que el valor recibido sea un entero:

```
{
  "ref_temperature": 25
}
```

| Origen | Destino | *Topic* |
|---|---|---|
| ***IoT Cloud*** | Lambda | temperature/reference/update |

## 4.5 Arquitectura de la solución

Una vez planteada la solución, ya se puede definir la arquitectura final, que permitirá realizar una simulación de una aplicación *Smart Home* de temperatura inteligente junto con una arquitectura *serverless IoT*.

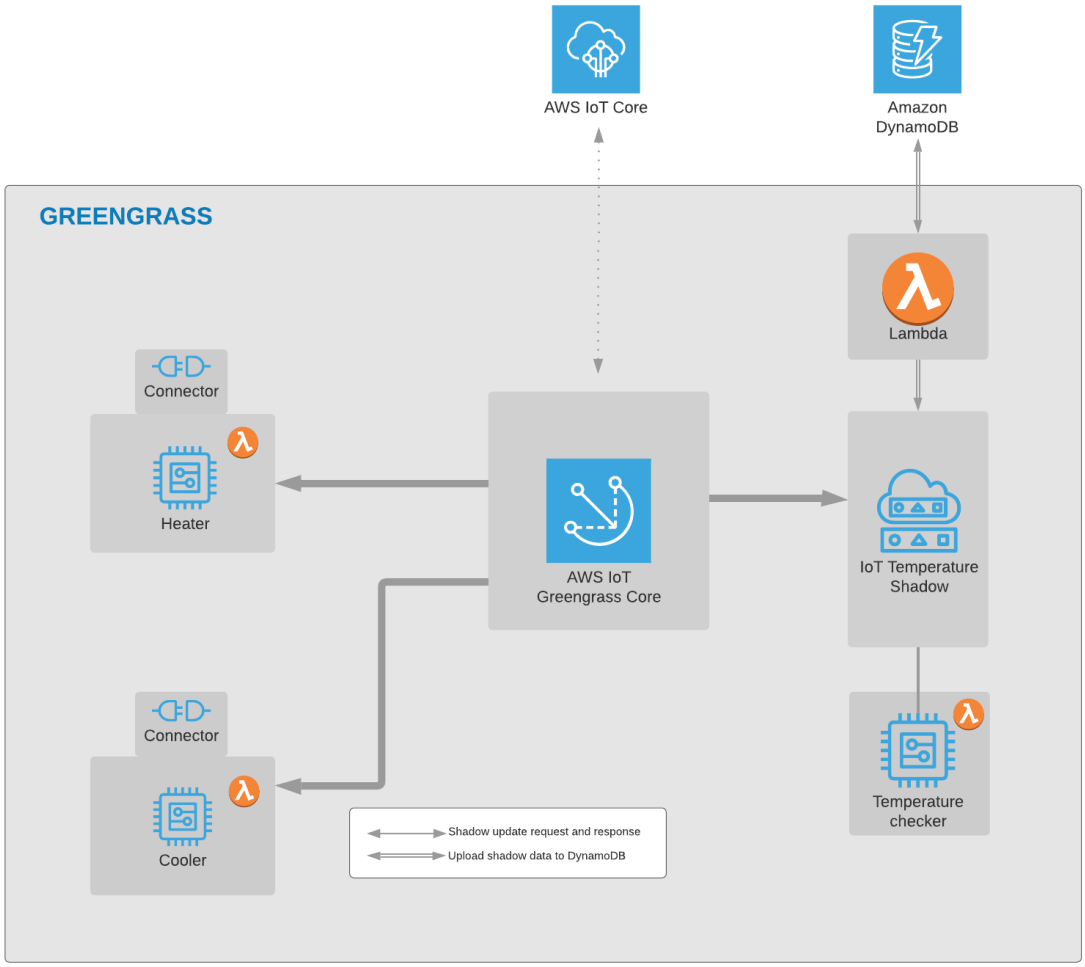



**Figura 28: Arquitectura de la solución**

En la Figura 28 se puede ver la arquitectura final de la solución con todos los elementos empleados para hacerla efectiva. Se pueden ver las interacciones de todos los distintos elementos empleando MQTT, excepto para la función Lambda con DynamoDB que usa HTTP.

De una manera sencilla se ha podido establecer una lógica de simulación de un regulador de temperatura. Para ello, se han podido emplear dispositivos como servicio mediante funciones *Lambda* que se suscriben a determinados *topics* y permite la interacción de todos los componentes. La sencillez de puesta en marcha y la capacidad de abstracción es un factor clave que demuestra la utilidad de *serverles* para soluciones *IoT*.

# 5 Validación tecnológica de la solución

Para validar la solución hay que tener en cuenta diversos aspectos, entre los que estaría comprobar que se ha llegado a una simulación de una solución funcional completa, la capacidad de replicar la solución, tiempo en aplicar actualizaciones, limitaciones, coste total de la solución y, por lo tanto, si se ha llegado a cumplir las características y los desafíos vistos en el apartado 3.

Lo primero es recordar cuales eran los objetivos de este trabajo para poder concluir si se ha llegado a una solución que los satisface:

## 5.1 Investigar los modelos *Cloud.*

Se ha estudiado todos los posibles modelos *Cloud* y entrado en detalle para poder entender que implicaciones tiene utilizarlo como parte de la solución final. Entre los modelos vistos se encuentra la base de la investigación de este proyecto, FaaS. Ha permitido profundizar con detalle las cualidades de este modelo de servicio *Cloud* y que ventajas tienen con respecto al resto.

## 5.2 Investigar las posibles arquitecturas *IoT.*

Para poder entender que nos puede aportar primero hay que conocer que tipos de arquitecturas *IoT* existen. Para el caso de una implementación *serverless IoT* es necesario utilizar *Edge computing* como la base de la arquitectura. Se han podido comprender las razones detrás de esto y, por lo tanto, continuar con el siguiente objetivo.

## 5.3 Investigar las arquitecturas *serverless* para soluciones *IoT.*

Se ha conseguido profundizar en detalle las posibles ventajas que aporta este tipo de arquitectura para *IoT.* Por lo que, una solución *IoT* se puede ver beneficiada debido a las cualidades innatas de las *FaaS* como *serverless*. Esto conlleva un nivel de abstracción que permite un despliegue, gestión y escalado de las soluciones *IoT* independientemente del nodo de computación *edge* y los dispositivos empleados.

## 5.4 Implementar un caso de uso con *serverless IoT.*

Se ha conseguido, con éxito, una implementación que simule un caso de uso real de *serverless IoT*. Esto ha permitido poder comprobar si este tipo de arquitecturas son realmente validas y que beneficios reales aportan

En concreto se ha simulado una aplicación de un regulador de temperatura que, mediante dispositivos de calefacción y climatización, permita ajustar la temperatura en base a una temperatura de referencia. Todo ello mediante dispositivos como servicio, utilizando los GPIO para poder observa una interacción en tiempo real, simulando el calefactor y climatizador con un par de leds. Un resumen de las interacciones del último día se pueden ver en la Figura 29.

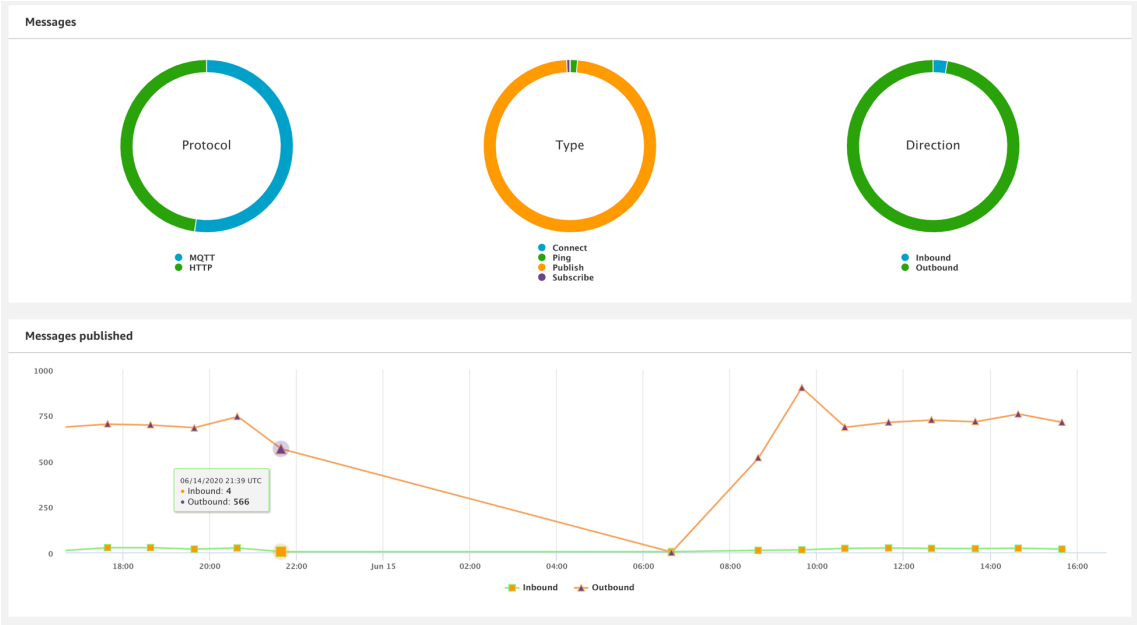



**Figura 29: Interacciones entre dispositivos**

Lo único que no se ha podido explorar, que es realmente uno de los puntos que mayor valor aporta una implementación *serverless* para *IoT*, sería la capacidad de aplicar modelos de *Machine Learning* que permitan mejorar los procesos y establecer un aprendizaje continuo. Esto se valora como un punto de mejora y entraría dentro del apartado de estudios futuros.

## 5.5 Validar la solución

### 5.5.1 Desafíos *Edge*

Para asegurar que se ha conseguido validar la solución hay que comprobar si esta solución consigue resolver todos, o parte, de los desafíos de *Edge Computing* que se usan como base para una arquitectura *serverless IoT.*

- **Agrupación de recursos y elasticidad rápida**:

Debido a que una implementación *serverless IoT* permite traer características propias de la *Cloud* también permite agrupar recursos y tener elasticidad. Esto se debe a que los dispositivos han de trabajar bajo el concepto de dispositivos como servicio y las cualidades de *FaaS.*

- **Seguridad**:

La tecnología usada para la implementación utilizada, que es AWS *IoT* Greengrass, permite establecer parámetros de seguridad que eviten problemas que puedan causar. A pesar de ello, el *stream* de datos a nivel de red local sigue sin estar encriptado. Esto se debe a que se asegura que el acceso a esa red es limitado, aún así es necesario que los dispositivos tengan certificados para poder darse de alta y establecer una comunicación.

- **Provisionado automático y gestión a escala**

Este es uno de los puntos que con mayor claridad se han cumplido, debido a que una de las funcionalidades que se puede encontrar dentro de la configuración de los Greengrass *Group* es el despliegue y provisión automático. Esto implica que no es necesario acceder al dispositvo Greengrass *Core* para poder desplegar nuevas versiones o provisionar nuevas interacciones con dispositivos.

Se puede ver un resumen de despliegues en la Figura 30, que además lleva un historial de versiones que permite redesplegar versiones anteriores. Esto demuestra las capacidades que tiene de reducir el *time-to-market* y las ventajas que ello conlleva.

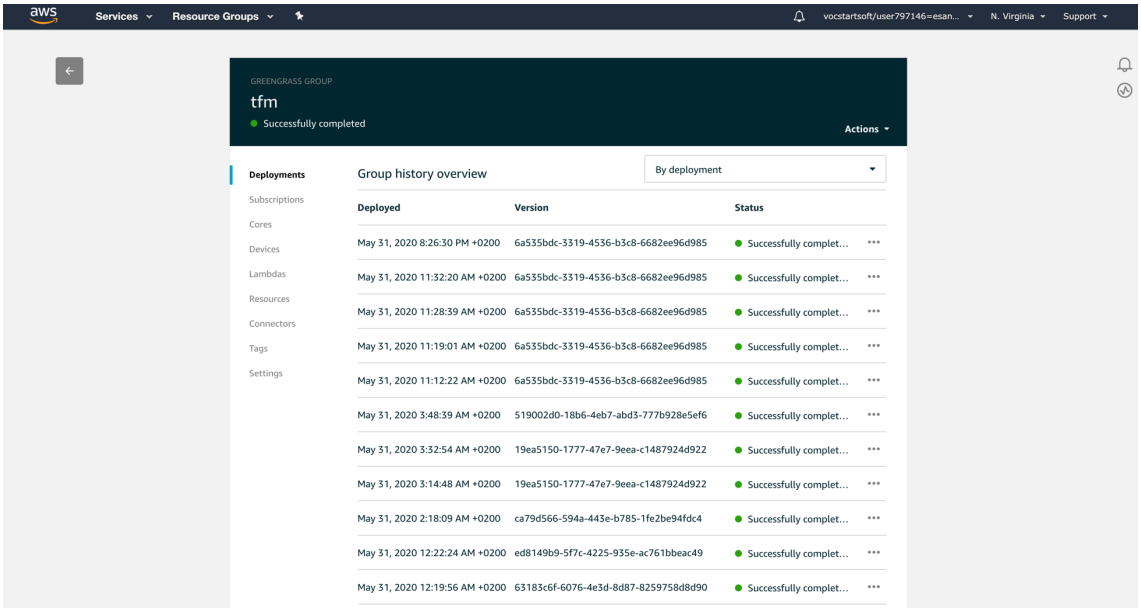



**Figura 30: Despliegues en Greengrass**

- **Programación de recursos *Edge* escasamente desacoplados y escasos**

Está tecnología se ha podido ver que permite un desacoplo de los recursos *Edge* debido a que permite ser desplegada y configurada en cualquier dispositivo con capacidades de computación suficientes. La instalación y configuración inicial ha de hacerse directamente en el dispositivo, pero se pueden establecer prácticas automáticas que permita olvidarse de ello.

- **Desarrollo de aplicación *deviceless***.

Debido a la naturaleza de los dispositivos no es posible un desacoplo completo de todos los elementos. Aún así, usando este tipo de arquitectura *serverless* se ha podido ver que permite abstraer los dispositivos hasta cierto punto, pudiendo personalizar las interacciones sin necesidad de tener que interactuar directamente con ellos. Por lo que toda la funcionalidad se hace directamente desde código y vía subscripciones.

- **Gobierno centrado en *Edge:***

Se ha podido ver que el tiempo de puesta en marcha de un dispositivo *Edge* no implica demasiada complejidad, por lo que la transición a este tipo de implementación no debería llevar mucho tiempo.

### 5.5.2 Características *Device as Services*

Recapitulando las características vistas en el apartado 3.2.1 se puede comprobar que se cumplen con la solución elegida de AWS *IoT* Greengrass. De hecho, describe a la perfección todas las características propias de AWS *IoT* Greengrass que se han podido ir viendo a lo largo en la implementación. Demostrando que la solución elegida es la ideal para poder validar una solución *serverles IoT.*

## 5.6 Limitaciones

La limitación principal, es que al usar una solución que depende de una plataforma *Cloud* obliga a los clientes a permanecer en esa *Cloud*, esto se conoce como vendor-lock-in. Por ello, hay que valorar las ventajas de utilizar está solución o si merece la pena buscar alternativas que puedan proporcionar características similares. Está claro que por el soporte, mantenimiento y evolución continua detrás de un servicio que depende de una plataforma *Cloud* puede contrarrestar este problema.

# 6 Conclusiones

El trabajo realizado ha permitido poder estudiar todos los conceptos relacionados con las tecnologías actuales, tanto para plataformas *Cloud*, como para soluciones *IoT*. El objetivo era centrar el trabajo alrededor de *FaaS* o *serverless,* para poder estudiar una nueva tecnología disruptiva que puede cambiar los modelos de negocio a medio y largo plazo.

Entre los modelos de negocio o posibles soluciones a implementar alrededor de esta tecnología, se encuentra una implementación *serverless IoT*. En el desarrollo de este proyecto se ha conseguido, con éxito, una implementación que simule un caso de uso real de *serverless IoT*. Esto ha permitido poder comprobar si este tipo de arquitecturas son realmente validas y que beneficios reales aportan

En concreto se ha simulado una aplicación de un regulador de temperatura que, mediante dispositivos de calefacción y climatización, permita ajustar la temperatura en base a una temperatura de referencia. Todo ello mediante dispositivos como servicio, utilizando los GPIO para poder observa una interacción en tiempo real, simulando el calefactor y climatizador con un par de leds.

Si algo hay que tener en cuenta han sido todas las complicaciones que han ido ocurriendo. La finalización del proyecto se ha retrasado una semana más de lo previsto por lo que la planificación no ha sido del todo correcta. Esto se debe, a que no se ha tenido en cuenta las posibles complicaciones que interrumpían el desarrollo continuo del trabajo. Por suerte, se han podido seguir los pasos que han permitido encontrar una tecnología adecuada y llegar a una solución óptima.

También, hay que valorar que se partía de la base de desconocimiento absoluto de las posibles tecnologías que podían permitir realizar un despliegue *serverless IoT*, por lo que parte del trabajo ha sido encontrar una que encajase con las características deseadas para validar la solución

Es necesario poder profundizar en las partes que no han podido ser exploradas con mayor detenimiento. La más importante es investigar el uso de *Machine Learning* junto con una tecnología *serverless* en la que, lamentablemente, no ha sido posible incluir la aplicación de modelos de *Machine Learning* integrados con la solución expuesta, pero al ser un factor muy relevante y diferenciador, es importante poder seguir explorándolo de cara a futuro.

La idea es poder utilizar los datos trasladados a la *Cloud* permitiendo optimizar los dispositivos de una aplicación *IoT*. Con ello, se puede mejorar los procesos, ahorrar costes innecesarios y estar tiempos de activación óptimos. Además, es posible entender patrones de uso que permitan ajustar o preparar los dispositivos analizando los datos.

# 7 Glosario

***IoT (Internet of Things)**:* Internet de las cosas

***IaaS (Infrastructure as a Service)**:* Infraestructura como servicio

***PaaS (Plataform as a Service)**:* Plataforma como servicio

***CaaS (Container as a Service)**:* Contenedores como servicio

***FaaS (Functions as a Service)**:* Funciones como servicio

***Capex (Capital Expenditure)**:* Inversión de capital

***Opex (Operating Expenditure)**:* Inversión de operación

***SLA (Service Level Agreement)**:* Acuerdos de nivel de servicio

***ROI (Return Of Inversión)**:* Retorno de inversión

***TTM (Time To Market)**:* Tiempo de salida al Mercado

***SOA (Service Oriented Architecture)**:* Arquitectura orientada a servicios

***DSL (Domain Specific Language)**:* Lenguaje específicio de dominio

***OS (Operative System)**:* Sistema operativo

***HTTP (Hipertext Transfer Protocol)**:* Inversión de operación

***API (Application Programming Interface)**:* Interfaz de programación de aplicación

***AWS (Amazon Web Service)**:* Servicios web de Amazon

***WoT (Web of Things)**:* Web de las cosas

***REST (Representation State Transfer)**:* Transferencia de representación de estado

***M2M (Machien to Machine)**: Máquina a máquina*

***TCP (Transport Control Protocol)**:* Protocolo de control de transporte

***UDP (User Datagram Protocol)**: Protocolo de diagrama de usuario*

***OSI (Open System Interconnection)**:* Interconexiones de sistema abierto

***MAC (Medium Access Control)**:* Control de acceso al medio

***WiFi (Wireless Fidelity)**:* Fidelidad sin cables

***BLE (Bluetooth Low Energy)**:* Bluetooth de baja energía

***IP (Internet Protocol)**:* Protocolo de internet

***NAT (Name Address Translation)**:* Tradución de nombre de dirección

***MTU (Maximum Transfer Unit)**:* Unidad de transferencia máxima

***LoWPAN (Low-power Wireless Personal Area Network)**:* Red de area personal de baja potencia

***CPU (Control Process Unit)**:* Unidad de control de procesos

***CoAP (Constrained Application Protocol)**:* Protocolo de aplicaicón límitado

***MQTT (Message Queuing Telemetry Transport)**:* Telemetría de transporte de mensajes de cola

***LLN (Low power and Lossy Networks)**:* Redes de baja potencia y con perdidas

***GPIO (General purpose input/output)**:* Entrada/salida de proposito general

***SDK (Software Development Kit)**:* Kit de desarrollo de *software*

# 8 Bibliografía


[1] N. B. RUPARELIA, Cloud Computing, MIT, 2016.

[2] M. Stigler, Beginning Serverless Computing, Apress, 2018.

[3] S. Cirani, G. Ferrari, M. Picone y L. Vetri, Internet of Things, Architectures, Protocols and Standards.

[4] P. Mell y T. Grance, The NIST Definition of Cloud Computing, National Institude of Standards and Technology (NIST), 2011.

[5] I. Melgrati, «Cloud services delivery models. Which can help your business?,» 14 06 2018. [En línea]. Available: https://imelgrat.me/cloud/cloud-services-models-help-business/. [Último acceso: 06 04 2020].

[6] N. Suryavanshi, «What are Cloud Computing Services [IaaS, CaaS, PaaS, FaaS, SaaS],» 8 11 2017. [En línea]. Available: https://medium.com/@nnilesh7756/what-are-cloud-computing-services-iaas-caas-paas-faas-saas-ac0f6022d36e. [Último acceso: 06 04 2020].

[7] M. Armbrus, A. Fox, R. Griffith, A. D. Joseph, R. Katz, A. Konwinski, G. Lee, D. Patterson, A. Rabkin, I. Stoica y M. Zaharia, Clearing the clouds away from the true potential and obstacles posed by this computing capability., 2010.

[8] P. SROCZKOWSKI, «Cloud: IaaS vs PaaS vs SaaS vs DaaS vs FaaS vs DBaaS,» [En línea]. Available: https://brainhub.eu/blog/cloud-architecture-saas-faas-xaas/.

[9] R. Borillo, «Qué es serverless y por qué adoptarlo en el desarrollo de tu próxima aplicación,» 01 04 2019. [En línea]. Available: https://www.genbeta.com/desarrollo/que-serverless-que-adoptarlo-desarrollo-tu-proxima-aplicacion.

[10] P. Sbarski, Serverless Architecture on AWS, Manning.

[11] C. Kidd, «FaaS vs Serverless: What's the Difference?,» 24 01 2019. [En línea]. Available: https://www.bmc.com/blogs/serverless-faas/.

[12] A. Ellis, 10 08 2017. [En línea]. Available: https://dzone.com/articles/introducing-functions-as-a-service-faas.

[13] P. Riti, Pro DevOps with Google Cloud Platform, Apress, 2018.

[14] AvSystems, «Benefits of a Cloud Platform in the IoT,» 24 09 2019. [En línea]. Available: https://www.avsystem.com/blog/iot-cloud-platform/.

[15] S. D. Jiang Rui, Architecture Design of the Internet of Things based on Cloud Computing, Qujing Normal University, 2015.

[16] Protego, «Serverless IoT,» [En línea]. Available: https://www.protego.io/serverless-security-for-iot/.

[17] K. Ismail, «Edge Computing vs. Fog Computing: What's the Difference?,» CMSWiRE, 14 08 18. [En línea]. Available: https://www.cmswire.com/information-management/edge-computing-vs-fog-computing-whats-the-difference/.

[18] AWS, AWS IoT Greengrass, Developer Guide.

[19] R. W. C. K. G. S. R. Fatih Bakir, Devices-as-Services: Rethinking Scalable Service Architectures for the Internet of Things.

[20] S. D. Stefan Nastic, Towards Deviceless Edge Computing: Challenges, Design

Aspects, and Models for Serverless Paradigm at the Edge.

[21] A. C. Gómez, «Diseño, desarrollo y prototipado de un Gateway M2M multiprotocolo para aplicaicones IoT,» 2018.

[22] R. L. Garrido, «Estudio Plataformas IoT,» 2015.

[23] «Raspberry Pi Documentation,» [En línea]. Available: https://www.raspberrypi.org/documentation/.

# 9 Anexos

## 9.1 Subscripciones

| Source | Target | Topic |
|---|---|---|
| Local Shadow Service | heater | $aws/things/Temperature_Sensor/shadow/update/accepted |
| heater | Local Shadow Service | $aws/things/Temperature_Sensor/shadow/update |
| Local Shadow Service | Temperature_Stats:Temperature_Stats | $aws/things/Temperature_Sensor/shadow/update/documents |
| IoT Cloud | Raspberry Pi GPIO:3 | gpio/tfm_Core/21/write |
| temperature_sensor:Temperature_Sensor | Raspberry Pi GPIO:3 | gpio/tfm_Core/21/read |
| Local Shadow Service | Temperature_Sensor | $aws/things/GG_TrafficLight/shadow/update/rejected |
| Raspberry Pi GPIO:3 | IoT Cloud | gpio/tfm_Core/21/state |
| Temperature_Sensor | Local Shadow Service | $aws/things/Temperature_Sensor/shadow/update |
| Local Shadow Service | Temperature_Sensor | $aws/things/GG_TrafficLight/shadow/update/accepted |
| Raspberry Pi GPIO:3 | heater_device:Heater_Device | gpio/tfm_Core/26/state |
| Local Shadow Service | Temperature_Sensor | $aws/things/Temperature_Sensor/shadow/update/delta |
| IoT Cloud | temperature_sensor:Temperature_Sensor | temperature/reference/update |
| cooler | Local Shadow Service | $aws/things/Temperature_Sensor/shadow/update |
| IoT Cloud | Raspberry Pi GPIO:3 | gpio/tfm_Core/26/write |
| Temperature_Sensor | Local Shadow Service | $aws/things/Temperature_Sensor/shadow/get |

| | | |
|---|---|---|
| temperature_sensor:Temperature_Sensor | Raspberry Pi GPIO:3 | gpio/tfm_Core/21/write |
| Local Shadow Service | cooler | $aws/things/Temperature_Sensor/shadow/update/rejected |
| Raspberry Pi GPIO:3 | cooler_device:Cooler_Device | gpio/tfm_Core/21/state |
| Local Shadow Service | Temperature_Sensor | $aws/things/Temperature_Sensor/shadow/get/accepted |
| temperature_sensor:Temperature_Sensor | Raspberry Pi GPIO:3 | gpio/tfm_Core/26/write |
| Local Shadow Service | heater | $aws/things/Temperature_Sensor/shadow/update/rejected |
| temperature_sensor:Temperature_Sensor | Raspberry Pi GPIO:3 | gpio/tfm_Core/26/read |
| Local Shadow Service | Temperature_Sensor | $aws/things/Temperature_Sensor/shadow/get/rejected |
| temperature_sensor:Temperature_Sensor | IoT Cloud | temperature/reference/state |
| Local Shadow Service | cooler | $aws/things/Temperature_Sensor/shadow/update/accepted |

## 9.2 Código

El código se puede encontrar en el siguiente enlace de github:

https://github.com/esanfru/tfm.git