UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño y despliegue de soluciones auto-adaptativas basadas en microservicios utilizando infraestructura elástica en la nube. Una aplicación sobre la IoT en la Industria 4.0

TRABAJO FIN DE MÁSTER

Máster Universitario en Ingeniería Informática

Autor: Carlos Izquierdo Martínez

Tutor: Joan Fons i Cors

Curso 2020 – 2021

Resumen

La Internet de las cosas (IoT) permite la posibilidad de adjudicar nuevas capacidades a objetos físicos y cotidianos. Es decir, permite dotarles de cierta inteligencia para que recojan datos de su entorno y puedan comunicarse, dando posibilidad a realizar nuevos escenarios y posibilidades con ellos.

Estas nuevas capacidades conectivas es lo que convierten a la IoT en una tecnología clave para la Industria 4.0 y su transformación de la industria tradicional en inteligente mediante la introducción de nuevos sistemas y servicios alimentados por datos que son capaces de actuar de forma autónoma.

Sin embargo, la aplicación de estos avances no es trivial y requiere de un procesamiento adecuado junto a una correcta comprensión y análisis de los datos generados. A esta problemática, se suman las complejas condiciones de funcionamiento con las que tienen que trabajar los sistemas a desarrollar que se enfrentarán a un entorno muy cambiante.

El proyecto aborda el problema del desarrollo de estos sistemas que deben de ser auto adaptables y responder ante cualquier necesidad del entorno aprovechando toda la información que tienen disponible.

Además, el proyecto evalúa las disposiciones actuales presentando arquitecturas basadas en microservicios como parte integral del desarrollo. Del mismo modo, se plantea la utilización de servicios de computación en la nube para la construcción, despliegue y monitorización automática de la infraestructura necesaria atendiendo a criterios de escalabilidad, eficiencia y alta disponibilidad.

Se muestra el diseño de una solución que cumple con los requisitos y se desarrolla un prototipo funcional para validar la propuesta.

Palabras clave: Internet de las cosas, Industria 4.0, Computación en la nube, Arquitectura de microservicios, Contenedores, Infraestructura como código, Computación autónoma, Sistemas auto-adaptativos

Resum

La Internet de les coses (IoT) ens permet la possibilitat d'adjudicar noves capacitats a objectes físics i quotidians. És a dir, ens permet dotar-los d'una certa intel·ligència perquè recullen dades del seu entorn i puguen comunicar-se, donant possibilitat a realitzar nous escenaris i possibilitats amb ells.

Aquestes noves capacitats connectives és el que converteixen a la IoT en una tecnologia clau per a la Indústria 4.0 i la seua transformació de la indústria tradicional en intel·ligent mitjançant la introducció de nous sistemes i serveis alimentats per dades que són capaços d'actuar de manera autònoma.

No obstant això, l'aplicació d'aquests avanços no és trivial i requereix d'un processament adequat al costat d'una correcta comprensió i anàlisi de les dades generades. A aquesta problemàtica, es sumen les complexes condicions de funcionament amb les quals han de treballar els sistemes a desenvolupar que s'enfrontaran a un entorn molt canviant.

El projecte aborda el problema del desenvolupament d'aquests sistemes que han de ser auto adaptables i respondre davant qualsevol necessitat de l'entorn aprofitant tota la informació que tenen disponible.

A més, el projecte avalua les disposicions actuals presentant arquitectures basades en microserveis com a part integral del desenvolupament. De la mateixa manera, es planteja la utilització de serveis de computació en el núvol per a la construcció, desplegament i monitoratge automàtic de la infraestructura necessària atenent criteris d'escalabilitat, eficiència i alta disponibilitat.

Es dissenyarà una solució que complisca amb els requisits i es desenvoluparà un prototip funcional per a validar la proposta.

Paraules clau: Internet de les coses, Indústria 4.0, Computació en el núvol, Arquitectura de microserveis, Contenidors, Infraestructura com a codi, Computació autònoma, Sistemes auto-adaptatius

Abstract

The Internet of Things (IoT) enable us to assign new capabilities to physical and everyday objects. In other words, it allows us to provide them with a certain intelligence so that they can collect data from their environment and communicate with each other, giving the possibility of developing new scenarios and possibilities with them.

These new connective capabilities are what make the IoT a key technology for Industry 4.0 and its transformation from traditional to smart industry by introducing new data-drive systems and services that are capable of acting autonomously.

However, the application of these advances is not trivial and requires adequate processing together with a correct understanding and analysis of the data generated. Added to this problem are the complex operating conditions with which the systems to be developed must work, as they will face a highly changing environment.

The project addresses the problem of developing these systems that must be self-adapting and respond to any need in the environment, taking advantage of all the information available to them.

Additionally, the project assesses current provisions by introducing microservices-based architectures as an integral part of development. Similarly, the use of cloud computing services is proposed for the construction, deployment and automatic monitoring of the necessary infrastructure according to criteria of scalability, efficiency and high availability.

A solution that meets the requirements will be designed and a functional prototype will be developed to validate the proposal.

Key words: Internet of things, Industry 4.0, Cloud Computing, Microservice architecture, Containers, Infrastructure as code, Autonomic computing, Self-adaptive systems

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Introducción

El número de dispositivos que están conectados a Internet no ha parado de crecer desde los últimos años. Diversos artículos especializados [1] apuntan que unos 25 mil millones de dispositivos han sido conectados en el año 2020 y han generado la suficiente cantidad de información como para poder monitorizarlos de manera continua, planificarles nuevas tareas, administrarlos a distancia o incluso, en algunos casos, programarlos para que tomen decisiones de forma autónoma.

La evolución de un conjunto muy amplio de tecnologías es quien ha permitido este crecimiento exponencial y con la incorporación de cada vez más diferentes tipos de sensores y actuadores en cualquier dispositivo físico, ya es posible recolectar la telemetría y los datos en tiempo real de su funcionamiento pudiendo ser reenviados a un controlador externo para que los evalúe e infiera una solución para dar una respuesta especifica al actuador.

Este intercambio de información entre sensores, controladores y actuadores también se ha visto reforzado por el surgimiento de nuevas tecnologías de localización, distribución, entrega de datos y conmutación de paquetes de entornos de red muy especializados y que cada vez están más extendidos como es el caso de la evolución [2] de lo que hoy se conoce como *Low Power Wide Area (LPWA)*. La gran diferencia de este tipo de redes con respecto a las tradicionales es su capacidad de permitir que dispositivos que disponen de una capacidad limitada de consumo de energía sean capaces de realizar comunicaciones de largo alcance de forma inalámbrica.

Además de estos avances en nuevos tipos de redes, también es necesario nuevos protocolos de Internet como *Constrained Application Protocol (CoAP)* o *Message Queuing Telemetry Transport (MQTT)* que utilizan arquitecturas software RESTful orientadas a la entrega de mensajes de forma asíncrona, para ayudar a estos dispositivos con capacidad limitada a poder establecer una comunicación desacoplada a pesar de sus difíciles condiciones de funcionamiento. Estos protocolos son capaces de que los dispositivos establezcan una comunicación directa y puedan tener un soporte mínimo de seguridad en algunos casos a pesar de tener una conectividad limitada y delimitada por su entorno que implique conexiones y desconexiones continuas.

El concepto que engloba este conjunto de tecnologías y pretende establecer un marco de referencia para el crecimiento de los dispositivos es el de la *Internet of Things (IoT)*. Existen multitud de definiciones para este nuevo concepto por diferentes autores especializados en la materia como se puede observar en la publicación que las recopila [3],

pero todas las opiniones coinciden en un punto común y es que permite la posibilidad de extender las capacidades de objetos físicos y cotidianos. Es decir, permite dotarles de una cierta automatización y toma de decisión para recoger datos de su entorno y poder comunicarse, dando posibilidad a realizar nuevos escenarios e implantaciones que abren caminos a nuevas innovaciones en distintos campos de investigación.

Estos objetos pueden tratarse de dispositivos ubicados en cualquier ámbito de la vida cotidiana, pero incluso también se pueden relacionar con equipos dentro de una industria. En relación con este punto, el autor del artículo [4] hace una definición de la IoT aplicada a la industria y lo define como «esos sistemas integrados dentro de máquinas que están conectados a Internet y tienen la capacidad de actualizarse y adaptarse a las necesidades bajo demanda». Partiendo de este punto, se desarrolla el concepto de que «proporcionan información de utilidad recolectada desde áreas geográficas muy distintas y remotas» y que con esta información es posible «diagnosticar los equipos, adelantarse a posibles interrupciones del servicio o fallos e interactuar con ellos de forma remota» con el objetivo de «hacerlos más eficientes y rentables para las empresas».

Los cambios producidos en este ámbito industrial donde se aplican éstas y otras tecnologías innovadoras son los que permiten mejorar los procesos de fabricación reduciendo sus costes y cambiando radicalmente los métodos de control y trazabilidad que hasta ahora se utilizaban en la industria tradicional, incluso pudiendo automatizar procesos y cambiarlos durante la marcha de forma autónoma. Estas nuevas capacidades conectivas que se requieren es lo que convierten a la IoT en una tecnología clave para la denominada Industria 4.0 y su objetivo de transformar la industria tradicional en inteligente.

La combinación de las tecnologías que componen la IoT con otros campos de investigación como la computación en la nube, la ciberseguridad, la fabricación aditiva o la automatización robótica, es un factor importante a tener en cuenta ya que puede garantizar el éxito de esta nueva revolución industrial y hacer posible una trasformación digital de las fábricas tal y como se puede ver respaldado por parte de diferentes especialistas en la publicación [5].

No obstante, la aplicación de estos avances supone un reto ya que el manejo de estos dispositivos que son capaces de generar mucha información requiere de un procesamiento adecuado junto a una correcta comprensión y análisis de los datos generados. A esta problemática, se debe tener en cuenta también el carácter distribuido que tienen muchos de los dispositivos y las complejas condiciones de funcionamiento con las que tienen que trabajar los sistemas a desarrollar que se enfrentan a un entorno muy cambiante donde los elementos están entrando y saliendo de forma constante.

Tal como se ha comentado anteriormente, muchas de las nuevas tecnologías han simplificado en gran medida la complejidad del desarrollo de estos sistemas y describen un soporte básico y una comunicación desacoplada entre dispositivos conectados que son capaces de adaptarse mínimamente a estas condiciones de funcionamiento tan cambiantes.

Otras tecnologías interesante que pueden encontrarse en la publicación [6], son los *Digital Twins* que describen un modelo digital que representa virtualmente a un elemento físico, dicho de una forma más sencilla, una copia virtual de un objeto físico que cuando se desacopla de la fuente original permite que los demás dispositivos pueden seguir co-

municándose con él y actualizar su estado. La tecnología permite que una vez el objeto físico recupera su conectividad, se sincroniza con su "gemelo digital" y actualiza su estado no afectando al funcionamiento general del sistema.

Sin embargo, la combinación de estas tecnologías que facilitan en gran medida el desarrollo de estas soluciones ya complejas de por sí trabajando bajo entornos muy cambiantes, son insuficientes cuando se aplican en algunos entornos como el industrial. En muchas ocasiones, cuando un servicio falla y su operativa está directamente relacionada a un equipo físico, soluciones como reiniciar el dispositivo para intentar recuperar la funcionalidad no funcionan y hay que buscar otras opciones.

El proyecto analizar y aborda la problemática del desarrollo de estas soluciones que necesariamente deben de ser auto adaptables y capaces de reconfigurarse y cambiar de arquitectura lógica si es necesario para responder ante cualquier necesidad del entorno aprovechando toda la información que tienen disponible.

1.1 Motivación

El trabajo lleva a cabo una investigación sobre una temática que se considera a simple vista compleja y que por lo tanto supone un reto como es el de abordar el desarrollo de una solución que sea capaz de reconfigurarse de manera autónoma en un entorno con difíciles condiciones de funcionamiento sin ninguna intervención más que la del propio sistema.

Existen diferentes tecnologías en el mercado que facilitan el desarrollo de este tipo de soluciones, pero cuando se aplica en un contexto industrial los resultados que arrojan se antojan insuficientes. Que una tecnología sea capaz de monitorizar un servicio y realizar acciones como reiniciarlo cuando se detecta una caída, en muchas ocasiones en el ámbito industrial no soluciona el problema ya que pueden influir otros factores.

Es necesario profundizar en el problema con más detalle y realizar una investigación de cómo puede desarrollarse una adaptación que permita a la propia solución encontrar la mejor solución y pueda aplicar reglas de reconfiguración que puedan pasar por desplegar nuevos servicios, retirar servicios no funcionales o incluso reconfigurar su arquitectura lógica.

El sistema debe poder aplicarse en un entorno real que demuestre la viabilidad y utilidad de llevar a cabo este tipo de soluciones que contribuyan a mejorar los procesos dentro de la industria. Demostrar que tener la capacidad de monitorizar métricas específicas en sistemas durante su tiempo de ejecución permite poder detectar y realizar previsiones heurísticas sobre anomalías que pudiesen darse, permitiendo realizar acciones proactivas y así no afectar a la disponibilidad y continuidad del servicio minimizando los tiempos entre fallos y recuperaciones. Como consecuencia directa resulta que los procesos en los que los equipos participan se tornan cualitativamente más eficientes y cuantitativamente más rentables para las empresas.

Se presentan las disposiciones actuales junto a una evaluación que delimita la implementación de la solución a utilizar arquitecturas basadas en microservicios como parte integral del desarrollo. Esta especificación en la arquitectura viene motivada principalmente por un interés personal con la intención de perfeccionar y afianzar la tecnología y, por otra parte, por la idoneidad con la problemática expuesta y la naturaleza distribuida que caracteriza la propia arquitectura.

Por otra parte, la agilidad que ofrece la propia arquitectura permite reutilizar funcionalidad ya desarrollada por terceros en algunas partes del proceso de implementación lo que permite centrarse en la cumplimentación de los objetivos.

Del mismo modo, se plantea la utilización de servicios de computación en la nube para la construcción, despliegue y monitorización automática de la infraestructura necesaria atendiendo a criterios de escalabilidad, eficiencia y alta disponibilidad. Con un enfoque profesional, el despliegue está intrínsecamente ligado a la utilización de herramientas de infraestructura como código con el objetivo de automatizar y permitir abstraer al sistema de la creación de infraestructura necesaria de cómputo y de red.

1.2 Objetivos

El objetivo principal del proyecto es abordar el problema de desarrollo de un sistema que debe de adaptarse de forma autónoma y responder ante cualquier necesidad del entorno aprovechando toda la información que tienen disponible.

Además, el proyecto evalúa las disposiciones actuales presentando arquitecturas basadas en microservicios como parte integral del desarrollo. Del mismo modo, se hace uso de los servicios de computación en la nube para la construcción, despliegue y monitorización automática de la infraestructura necesaria atendiendo a criterios de escalabilidad, eficiencia y alta disponibilidad.

Se definen los siguientes hitos que van a determinar si los objetivos son alcanzables, razonables y medibles en el tiempo:

- Desarrollar una solución modelo y fácilmente extensible utilizando una arquitectura basada en microservicios que utilice patrones de comunicación distribuida.
- Empaquetar el desarrollo en contenedores y crear una imagen por cada servicio que compone la solución.
- Levantar la solución en una infraestructura de nube pública con una configuración de gestión básica.
- Automatizar el despliegue en la nube utilizando una herramientas de *Infraestructure as Code* (IaC).
- Proporcionar los mecanismos o componentes necesarios de adaptación para que los servicios se reconfiguren automáticamente modificando su comportamiento sin alterar su implementación base.
- Desarrollar escenarios personalizados e implementar pruebas que demuestren la viabilidad y utilidad de llevar a cabo este tipo de soluciones en entornos reales.

1.3 Impacto Esperado

Tal como se ha comentado anteriormente, los Digital Twins son aquellas tecnologías que permiten crear un modelo digital que representa virtualmente a un dispositivo físico, dicho de otra forma, se trata de una copia digital de un objeto físico que está continuamente sincronizando su estado. Uno de los principales objetivos de esta tecnología es mantener el estado y seguir ofreciendo una disponibilidad virtual al resto de servicios incluso en el caso de una caída o interrupción del objeto físico.

Esta tecnología esta muy relacionada con la IoT, ya que su crecimiento permite acelerar y ayudar al desarrollo de los Digital Twins gracias a las similitudes de ambas tecnologías y su naturaleza conectiva común. Y es justo en este campo donde más aporta la IoT cuando se relaciona con estos modelos de representación virtual, su conectividad, ya que permite transformar y cambiar la forma de interacción entre los Digital Twins y los dispositivos físicos.

Los principales proveedores que proporcionan servicios de computación en la nube ofrecen una implementación de este tipo de tecnologías a través de un modelo de pago por uso a los usuarios, pero, por detalles que se describirán y analizarán en más profundidad en el capítulo del estado del arte, se puede adelantar que esta tecnología a simple vista es insuficiente para resolver el problema de adaptación y reconfiguración que se plantea a lo largo de todo el proyecto.

Además, se debe tener en cuenta que la utilización de estos servicios implica una elevada dependencia en el proveedor de servicios *cloud* al tener que adaptar la implementación y programación de los dispositivos a los recursos proporcionados. Este concepto se conoce dentro de la computación en la nube como vendor lock-in y sus repercusiones han sido objeto de estudio por diferentes especialistas en la tecnología en diversas publicaciones [7], donde se resalta la complejidad de trasladar los recursos a otro proveedor una vez están configurados.

La solución desarrollada debe de ser genérica y capaz de poder adaptarse y extender sin dificultad a cualquier situación y ámbito posible. Es importante destacar que la infraestructura desplegada para el funcionamiento de la solución debe de ser trasparente e independiente para los servicios involucrados y únicamente conocida para el componente de adaptación.

De esta forma, el componente de adaptación dentro del sistema autoadaptable que se analiza a lo largo de todo el proyecto es el que tendrá la capacidad de decidir sobre donde ubicar la infraestructura y que servicios genéricos será necesario desplegar.

1.4 Metodología de trabajo

El proyecto parte de una situación con un alto nivel de complejidad debido a que se está tratando un problema que todavía no ha sido resuelto e implica la utilización y aprendizaje de nuevas tecnologías disruptivas junto a un análisis profundo que pretende alcanzar a una propuesta de solución innovadora.

Una combinación de la metodología tradicional de investigación junto a la flexibilidad que proporciona una metodología ágil resulta de especial interés en este proyecto, donde la iteración entre diferentes puntos y la obtención de conclusiones a lo largo de todo el proyecto permiten replantear y adaptar el proyecto a las circunstancias sin dejar de tener presente los objetivos.

La planificación del proyecto consta de siete fases que se detallan a continuación.

- **Fase 1. Selección y delimitación de la temática:**

  Se decide y selecciona de entre varias opciones cual va a ser la línea de investigación que se va a desarrollar a lo largo del proyecto, dejando claro los límites y aspiraciones a conseguir con el objetivo de tener claro en que se va a trabajar.

- **Fase 2. Planteamiento inicial del problema:**

  A raíz de la motivación que permite establecer una temática sobre la que desenvolverse surge la problemática que se pretende solventar. Este planteamiento inicial pretende establecer si se trata de un problema que aún no tiene solución conocida, si tiene una solución parcial o si existe no funciona como se espera que funcione.

- **Fase 3. Marco de investigación y análisis:**

  Concentra la investigación principal de referencias y trabajos anteriormente desarrollados en la temática para conocer los diferentes puntos de vista alcanzados en sus conclusiones y poder establecer una posición de partida. Si se encuentran trabajos que traten de forma explícita la problemática planteada inicialmente, se analiza su funcionamiento para reorientar el planteamiento inicial.

- **Fase 4. Aprendizaje de nuevas tecnologías potenciales:**

  Los antecedes descubiertos en el punto anterior no solo permiten tener un contexto más detallado sobre el problema a abordar, sino que también permiten descubrir nuevas tecnologías que se han utilizado durante su desarrollo, de las cuales es necesario realizar un aprendizaje para poder también utilizarlas si son adecuadas.

- **Fase 5. Análisis detallado y justificación del problema:**

  Durante esta fase se analiza el problema de forma más detallada con un contexto tecnológico aprendido y el conocimiento de artículos de investigación y trabajos sobre la misma temática. De esta forma, es posible afrontar el problema con toda la información e identificar nuevas oportunidades de mejora.

- **Fase 6. Diseño e implementación de la solución:**

  Empieza con la definición conceptual de un sistema y se va iterando sobre diferentes puntos clave ya definidos anteriormente hasta llegar a una propuesta de solución. La solución se implementa como prueba conceptual utilizando métodos y tecnologías que han sido aprendidos en fases anteriores.

- **Fase 7. Pruebas funciones, resultados y conclusiones:**

  Se diseñan una serie de pruebas sobre la solución desarrollada con el objetivo de extraer información y resultados que permitan asegurar que se han cumplido los objetivos definidos al principio del proyecto.

Se puede ver una estimación del tiempo empelado y un desglose de las tareas realizadas en el diagrama de Gantt que se muestra a continuación.

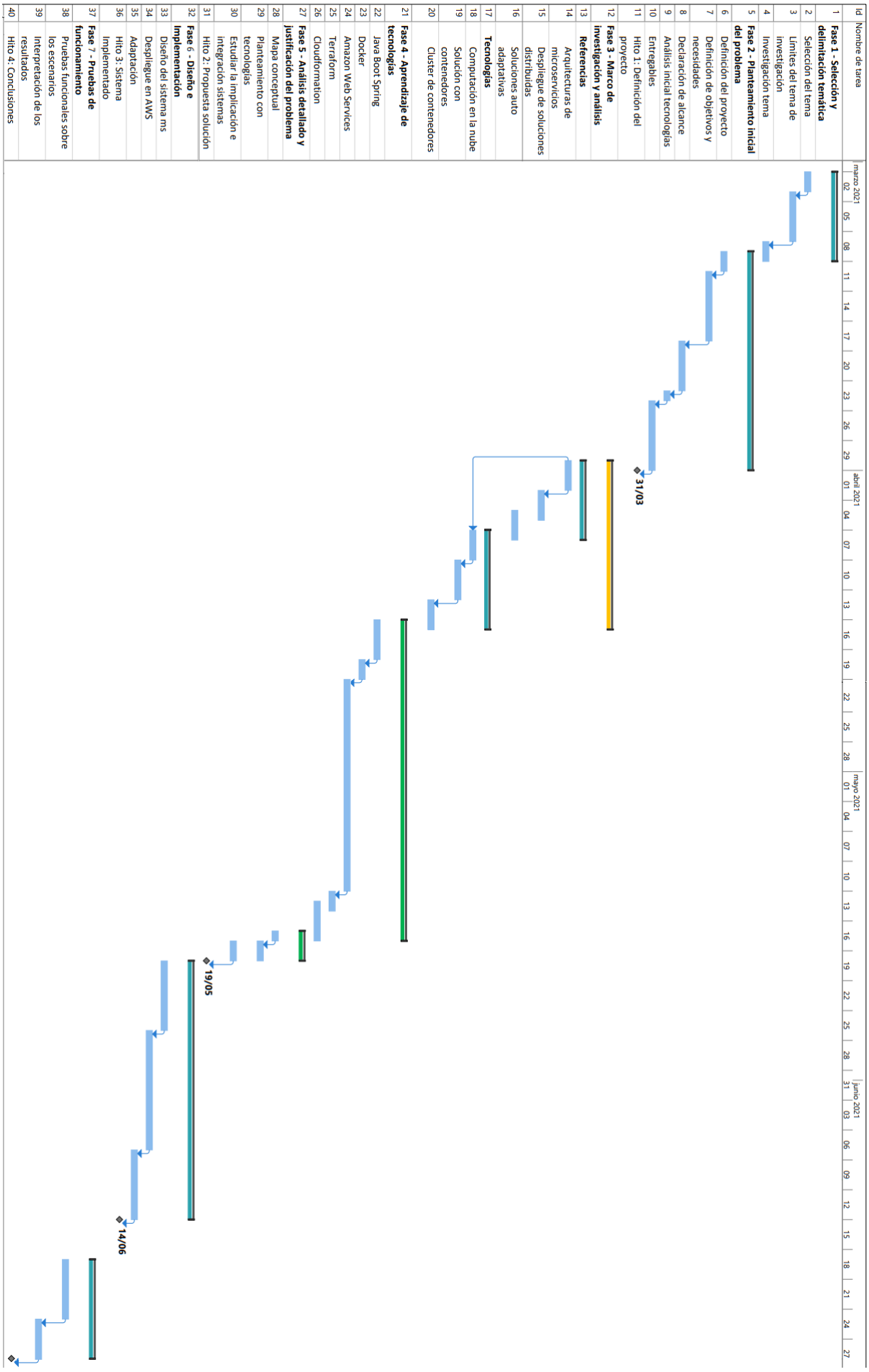


Figura 1.1: Diagrama de Gantt del proyecto

1.5 Estructura de la memoria

El siguiente apartado detalla la estructura general del documento que está compuesto por un total de siete capítulos y un anexo. En cada uno de ellos, se describe su contenido y se hace un recorrido rápido por sus diferentes apartados.

En el **capítulo 1 (Introducción)**, se expone de una manera sencilla el problema que se aborda a lo largo de todo el proyecto con la intención de introducir al lector dentro de un contexto conocido. La introducción trata la complejidad que supone el desarrollo de una solución para la Internet de la cosas y la Industria 4.0 situados dentro de un entorno cambiante e inestable.

A lo largo de sus diferentes apartados se plantean aspectos como la motivación del proyecto, el establecimiento de objetivos que definen que se pretende conseguir, el impacto esperado por parte de la solución o la metodología utilizada. También se puede encontrar otros aspectos relacionados con este documento como su estructura y organización o convenciones de texto.

En el **capítulo 2 (*Estado del arte*)**, se explica de forma detallada el análisis llevado a cabo cuando se afronta el desarrollo de una solución que necesariamente tiene que reconfigurarse o adaptarse al entorno cambiante. Se estudian diferentes referencias, documentos técnicos y alternativas con el objetivo de analizar y disponer de toda la información para la selección de potenciales tecnologías y métodos de desarrollo a utilizar.

Dentro de los diferentes apartado se crítica el contexto tecnológico al contemplar las limitaciones de disponen las soluciones y plataformas existentes, con la intención de identificar las oportunidades de mejora y los aspectos fundamentales a los cuales no se presta funcionalidad y pueden tener cabida dentro del contexto del proyecto definiendo por tanto también una propuesta tecnología diferencial.

En el **capítulo 3 (Análisis del problema)**, se describe la utilización de diferentes técnicas y modelados conceptuales para hacer una representación de un sistema base que se expone directamente al entorno cambiante y se observan sus resultados. De esta forma, al utilizar conceptos sencillos y básicos en la representación del sistema, se facilita la comprensión y la simulación de los problemas de adaptación a los que se puede enfrentar.

A partir del análisis del problema realizado, se identifica en el siguiente apartado las múltiples soluciones posibles al problema planteado de acuerdo con las diferentes interpretaciones de los resultados.

En el **capítulo 4 (Diseño de la solución)**, se presenta la solución final escogida de entre las posibles y atendiendo a los criterios de selección definidos en el capítulo anterior, detalla con más profundidad en que consiste y estructura como se va a implantar y validar mediante pruebas.

El apartado del plan de trabajo describe las fases de desarrollo de la solución y define una planificación y división de tareas con una estimación de esfuerzos. A lo largo del resto de apartados, se plantean aspectos como la identificación de requisitos, diseño de la solución, arquitectura del sistema y sus diferentes componentes, tecnologías utilizadas y se desatacan los diferentes problemas, dificultades y desviaciones encontrados durante el desarrollo.

En el **capítulo 5 (Implementación y despliegue)**, se detalla la instalación y puesta en marcha en un entorno real del sistema desarrollado bajo los escenarios definidos en el

que el entorno es configurado de forma que se satisfagan las condiciones necesarias para ser probado.

En el apartado de pruebas se describen todas las pruebas realizadas para verificar que la solución es funcional y capaz de adaptarse de manera autónoma al entorno y reconfigurarse a nivel de arquitectura incluso en un entorno elástico de un proveedor *cloud*. El apartado de resultado presenta los datos medibles y los relaciona con los objetivos definidos en el capítulo introductorio.

En el **capítulo 6 (Conclusiones)**, se reflexiona y relaciona la interpretación de los resultados y las pruebas con los objetivos definidos inicialmente para hacer una valoración final que manifiesta si se han alcanzado todos los medibles y si han surgido nuevos objetivos durante el desarrollo.

Así mismo, en uno de los apartados se compara el dominio de las tecnologías antes y después del proyecto con una representación sobre la evolución del aprendizaje y la concordancia con las competencias adquiridas en los estudios cursados.

El apartado de trabajos futuros presenta líneas de desarrollo alternativas que se abren tras la aplicación de la solución y sus resultados, así como una lista de tareas pendientes que no se han desarrollado por estar fuera del contexto del proyecto pero que pueden aportar funcionalidades interesantes para su estudio y desarrollo.

La **Bibliografía**, muestra una lista de fuentes de conocimiento que se han escogido como referencia y entre los que predominan artículos de investigación para reafirmar y construir el trabajo desarrollado. Todas las referencias aparecidas se utilizan a lo largo de este documento identificadas mediante el número correspondiente.

Finalmente, se adjunta un **siglario** y **apéndice** que muestra algunos aspectos conceptuales representativos del despliegue de y puesta en marcha de la solución para su referencia.

1.6 Convenciones

El proyecto sigue una normativa de marcado a lo largo de todo el documento y por tanto existen unas definiciones y terminologías específicas relevantes que están asociados a palabras o frases que es conveniente destacar.

- Los anglicismos se remarcan en cursiva.
- Se usan comillas angulares españolas « » para las citas textuales.
- Algunos términos de carácter técnico y muy específico de la temática están recogidos mediante siglas. Estas siglas vienen definidas en su apartado correspondiente al final del documento para fines de consulta.
- El código fuente se muestra remarcado en cursiva y con una tipología de fuente Courier. Solo este tipo de contenido es el que emplea esta tipología.

CAPÍTULO 2
Estado del arte

El estado del arte permite representar, detallar y analizar las tecnologías claves, investigaciones y publicaciones que se han realizado en torno al ámbito introducido en los primeros capítulos del proyecto. La intención es la de sintetizar y destacar los aspectos más relevante de lo que se ha estudiado y presentar los conocimientos que existen alrededor del desarrollo de estos sistemas y soluciones aplicados a la industria con el objetivo de establecer un punto de partida.

2.1 Plataformas IoT

Como se ha adelantado en el capítulo anterior, los principales proveedores que proporcionan servicios de computación en la nube ofrecen una solución propia de Platform as a Service (PaaS) para dispositivos IoT con un modelo de facturación de pago por uso. Este tipo de plataformas ofrecen una implementación todo en uno para conectar dispositivos y comunicarse con ellos a través de recursos propios como APIs y colas de mensajerías. Algunas de las principales empresas que ofrecen estos servicios son Amazon con la solución AWS IoT Core, Microsoft por su parte con Azure IoT Suite y por otra parte, es posible encontrar otras plataformas especializadas como DeviceHive.

Lo interesantes de estas plataformas es que proporcionan una implementación de la tecnología ya definida anteriormente como Digital Twins que permitía crear un modelo digital para representar virtualmente a un dispositivo físico. Ésta copia digital de un objeto físico está continuamente sincronizando su estado y su configuración consiste en seleccionar los aspectos funcionales y características de los que se quiere mantener un gemelo y sean imprescindibles para seguir ofreciendo una disponibilidad virtual al resto de servicios incluso en el caso de una caída del objeto físico. En Amazon Web Services (AWS) ofrecen un producto preparado que cubre estos aspectos bajo la denominación Device Shadow mientras que el resto de las plataformas mantiene la nomenclatura de Digital Twin.

Implementar esta u otras de las múltiples funcionalidades ofrecidas por estas plataformas implica adaptar las comunicaciones de los dispositivos a la nube y el formato de los mensajes al referenciado en la documentación de la plataforma. Además, se requiere que los dispositivos o sus abstracciones tengan un control mínimo sobre políticas de seguridad y gestión de certificados asociados a la plataforma y en ocasiones esto no es posible garantizarlo debido a que pueden encontrarse dispositivos que no pueden soportar esta carga extra debido a sus limitaciones en cuanto a procesamiento y/o computo.

En estos casos, para integrarlos dentro de éstas plataformas se requeriría depender de controladores externos que realicen estas gestiones por ellos y puede llegar a aumentar enormemente la complejidad global de la solución.

De forma adicional, existe la problemática asociada a como sobre los dispositivos nativos existe una fragmentación a nivel de modelo y características, que se ve acrecentada por la ausencia de interoperabilidad de diversos fabricantes para la realización de operaciones nativas. No obstantes, la solución de intercomunicación tradicional de usar protocolos estandarizados comunes sigue sin disipar la complejidad y coste de mantenimiento.

Cuando se hace uso de estos recursos también hay que tener en cuenta que no se tiene el control total sobre la infraestructura donde se están ejecutando las copias virtuales y se genera una elevada dependencia con el proveedor *cloud* al tener que adaptar la implementación y programación a las APIs, recursos y librerías personalizadas de la plataforma. Este vendor lock-in, imposibilitará en un futuro trasladar los recursos a otro proveedor una vez están configurados sin tener que volver a implementar las comunicaciones desde cero y al mismo tiempo limita la forma en la que la plataforma puede evolucionar en características, ya que depende en exclusiva de que los componentes subyacentes sean compatibles.

Sin embargo, no es el único problema encontrado ya que no suele ser habitual el permitir por parte del proveedor extender su funcionalidad con soluciones propias. Esto impide poder abordar el problema de adaptación y reconfiguración a nivel de arquitectura que se está buscando y persiguiendo a lo largo del proyecto ya que cuando un servicio falla y su operativa está directamente relacionada a un equipo físico, no es posible ni eficiente la respuesta que ofrece la plataforma de solucionar el problema reiniciando el servicio, sino que hay que buscar otras opciones.

La decisión tomada es la de desarrollar una solución genérica que sea capaz de adaptarse sin complejidad y abstraer su funcionamiento para poder desplegarse en cualquier infraestructura realizando los mínimos cambios posibles y que sean trasparentes de cara a los servicios y dispositivos.

A continuación, se puede ver en la tabla 2.1 una comparación de las características más relevantes que se han detectado en las principales plataformas IoT existentes.

Plataforma IoT	Comunciación	Servicios Destacados	Desarrollar Funcionalidad
Amazon Web Services IoT Core	HTTP MQTT	Autenticación Entorno de desarrollo	Si, pero con framework propio
Microsoft Azure IoT Hub	HTTP MQTT AMQP	Autenticación Monitorización Gestión remota	No
Google Cloud IoT Core	MQTT	Monitorización	No
Solución generica	HTTP MQTT	Despliegue en cualquier sitio	Si

Tabla 2.1: Comparación de servicios y caracterísitcas ofrecidos en las plataformas IoT existentes

2.2 Arquitecturas de aplicación

Los dispositivos dentro de una solución IoT, como se ha adelantado anteriormente, pueden llegar a tener ciertas limitaciones relacionadas con el consumo de energía, capacidad de procesamiento, almacenamiento o conectividad. Algunos autores han analizado esta situación en diferentes artículos [8], donde se llega a la conclusión de que estás complicadas condiciones de funcionamiento implican que la implementación de un sistema o plataforma que controle estos dispositivos no puede hacerse mediante el uso arquitecturas tradicionales de desarrollo. Artículos [9], defienden y desaconsejan el uso de patrones monolíticos debido a la gran variedad de tecnologías que se integran para el desarrollo de estas soluciones.

Las arquitecturas tradicionales monolíticas son aquellas donde se concentran todos los aspectos funcionales de una solución bajo una misma aplicación que con frecuencia concentran todos los elementos bajo un mismo punto crítico de fallo. Estas arquitecturas no disponen de una gran adaptabilidad o escalabilidad cuando concentran un gran número de tecnologías y su desarrollo se vuelve complejo cuando se requiere añadir una nueva característica o cambiar una existente donde se tiene que garantizar la compatibilidad con el resto de las funcionalidades y asegurar que el servicio común continúe funcionando como se espera.

Otros inconvenientes tal como se analiza en [10], ocurren cuando es necesario realizar un despliegue de una actualización en un entorno de producción. El autor manifiesta que el comportamiento normal en el despliegue implicará que todos los servicios deban de reiniciarse causando una interrupción para los usuarios y va más allá cuando detalla que, si una de las funcionalidades falla debido a un error durante su ejecución, el resto de los servicios quedarán inoperativos.

Una solución con dispositivos IoT requiere que la solución sea capaz de proporcionar flexibilidad, escalabilidad y disponibilidad en tiempo real debido a la naturaleza distribuida de la amplia gama de dispositivos que es posible conectar, tal y como sostiene el autor del artículo [11]. Además, se requiere de mecanismos que permitan el descubrimiento de estos dispositivos y realizar un correcto tratamiento de la información que se genera y recibe en todo momento.

El autor del artículo [12] reafirma lo anterior y vuelve a manifestar que uno de los aspectos claves a tener en cuenta durante el desarrollo, es conseguir una buena interoperabilidad entre los diferentes servicios y aplicaciones que se ejecutan en los dispositivos. De entre las diferentes opciones posibles, se considera que la más interesante es la de combinar una arquitectura basada en microservicios junto con una Interfaz de programación de aplicaciones (API) o una implementación de colas de mensajes (Queue service), que esté bien definida para cualquiera de los dos casos y que permita una interacción ya sea basada en operaciones o intercambio de mensajes entre los recursos.

Una arquitectura de microservicios propone la idea de distribuir las aplicaciones en un conjunto de servicios, en los que cada uno es independiente de los demás y proporciona una funcionalidad de la solución. Un microservicio representa una funcionalidad que interactúa a través de mensajes con otros microservicios y se centra en un aspecto único que puede ser reutilizado reduciendo los costes de desarrollo y mantenimiento.

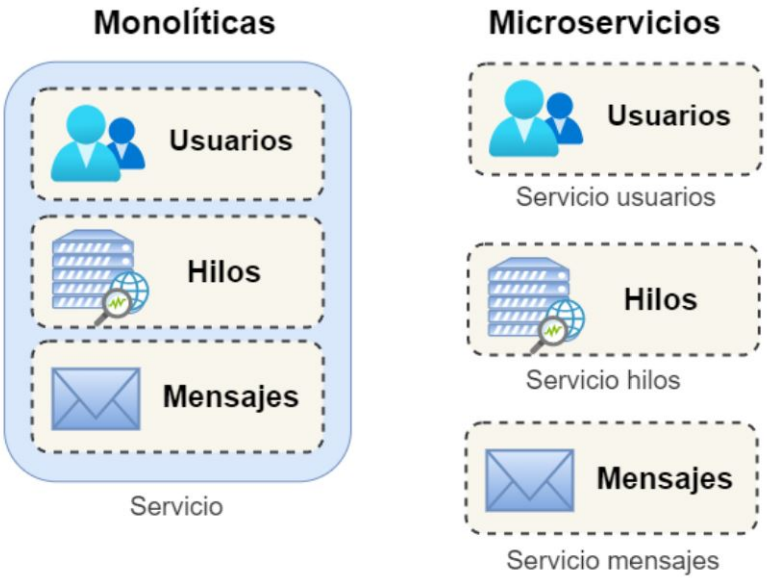


Figura 2.1: Arquitectura monolítica y basada en microservicios

Los microservicios han sido objeto de interés y estudio por parte de la comunidad científica los últimos años y en especial, su aplicación a los actuales sistemas de fabricación en la industria según recogen algunos artículos [13] [14]. Éste último describe una solución que utiliza este tipo de arquitecturas para la implementación de un *middleware* de comunicaciones que se utiliza para automatizar diferentes procesos en el proceso transformación digital de una fábrica.

Otros recogen [15], la utilización de microservicios y comunicaciones a través de MQTT y Representational state transfer (REST) para proponer una plataforma colaborativa de la Industria 4.0 que posibilite una monitorización en tiempo real, una optimización automática y una negociación de dispositivos IoT integrados en máquinas de producción con el objetivo de mejorar los procesos de diferentes áreas como la fabricación o la cadena de suministro.

Cuando se profundiza y se investiga más sobre cómo afrontar el desarrollo y el tipo de arquitectura a utilizar para tomar un punto de referencia, se puede ver que la que está basada en microservicios es la predominante entre multitud de artículos técnicos para este tipo de soluciones con dispositivos distribuidos y se entiende porque resulta una tecnología clave.

El capítulo introductorio ya había planteado la utilización de este tipo de arquitectura basada en microservicios por una motivación personal y de aprendizaje por parte del autor, pero durante esta tapa de investigación se ha podido reafirmar como la mejor opción dada la idoneidad con la problemática expuesta y la propia naturaleza distribuida de la arquitectura.

2.3 Virtualización, contenedores y despliegue automatizado

Uno de los autores en [16] reflexiona sobre las diferentes formas de poner en marcha una solución de monitorización que ha desarrollado en su proyecto y cual es la mejor forma de ubicar dentro de su infraestructura los diferentes microservicios que ha implementado. De entre las múltiples opciones que analiza, destaca y justifica que lo mejor es utilizar contenedores a la hora de realizar el despliegue, relacionando el alto potencial que le proporciona esta tecnología para poder crear un escenario completamente automatizado y que le permite reducir la complejidad del proyecto.

Un contenedor es un paquete que incluye todo lo necesario para que una aplicación pueda ejecutarse de forma rápida y unitaria en un entorno de cómputo. Esto incluye todas las herramientas necesarias para su ejecución como pueden ser dependencias, librerías y algunos archivos binarios. A diferencia de virtualización tradicional, se utiliza un aislamiento y serialización de llamadas a sistema donde todos los contenedores comparten el núcleo del sistema operativo, pero manteniendo un espacio almacenamiento propio.

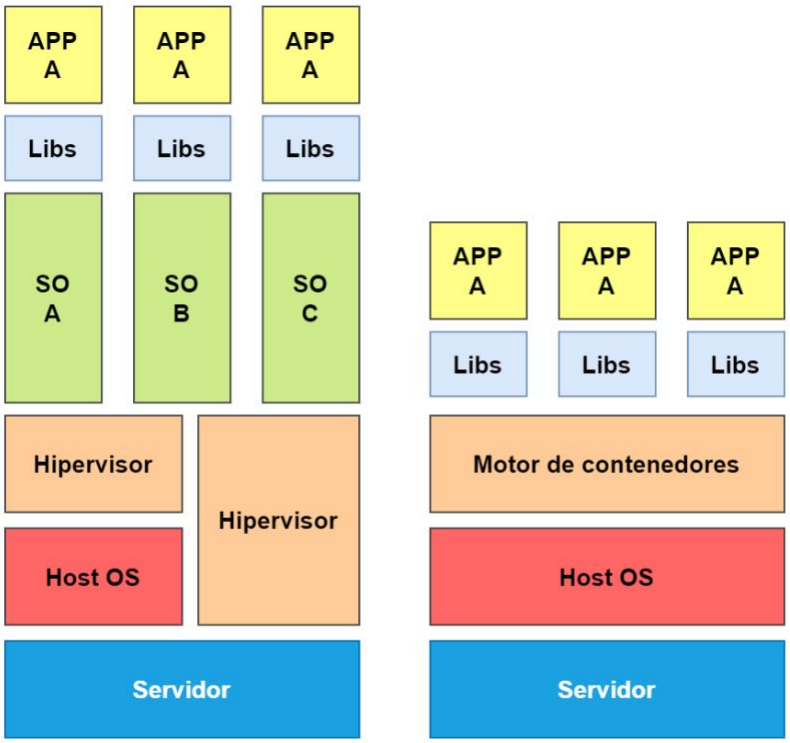


Figura 2.2: Virtualización tradicional con hipervisores y con contenedores

Las soluciones que utilizan este tipo de aislamiento y en concreto las que están basadas en contenedores, están creciendo rápidamente entre el desarrollo de soluciones IoT tal como recoge [17]. Una de las ventajas más destacadas y el principal motivo de elección en comparación con las tradiciones tecnologías de virtualización basadas en máquinas virtuales, es la facilidad que ofrecen de poder aprovisionar de forma rápida los microservicios utilizando unos recursos mínimos y únicamente los estrictamente necesarios junto con una política de consumo de energía reducido y la posibilidad de desplegarlos en cualquier tipo de infraestructura.

	Maquina Virtual	Contenedor
Virtualización	Hipervisores	Motor de contenedores
Funcionamiento	MV comparten kernel/nucleo hipervisor	Grupo aislados de procesos comparten kernel y SO
Aislamiento	Completo	Espacio de nombres
Despliegue	Normal	Muy rápido
Aprovisionamiento de recursos	Normal	Muy bajo
Escalabilidad	Normal	Muy rápida
Configuración de entornos y dependencias de apps	Complejo	Muy fácil

Tabla 2.2: Comparación entre maquinas virtuales y contenedores

El autor de [18] también destaca otro aspecto clave como es la capacidad de gestionar y organizar varios contenedores a la vez, sobre todo, enfocado a la hora de facilitar una rápida distribución de servicios idénticos para atender necesidades de disponibilidad o responder ante la incertidumbre en el estado de los recursos si estos están asociados a dispositivos físicos. Estos despliegues y la posibilidad de replicar más de un servicio en diferentes configuraciones permiten mejorar la respuesta de los microservicios en sus tiempos de ejecución, latencia o disponibilidad.

Docker es una de las implementaciones de esta tecnología de contenedores donde sus imágenes son los componentes básicos a partir de los cuales se lanzan las aplicaciones. Su proyecto de *software* libre para la automatización y despliegue de contenedores es uno de los más extendidos y utilizado en el desarrollo de soluciones *software*, incluso descrito como un estándar de facto en la actualidad tal como algunos autores publican [18]. Su funcionalidad se ha extendido con el paso del tiempo hasta llegar al punto donde es incluso posible aplicar diferentes estrategias de planificación inteligentes que proporcionan mucha flexibilidad en la distribución de estos contenedores.

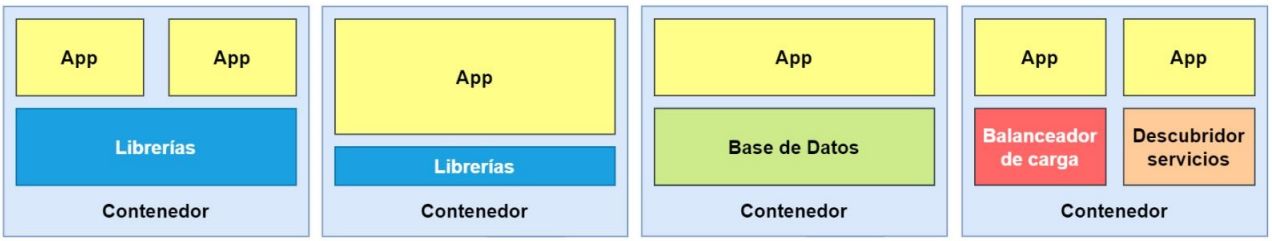


Figura 2.3: Arquitecturas y configuraciones de aplicaciones basadas en contenedores

2.4 Computación en la nube

El planteamiento de utilizar en la proyecto tecnologías de contenedores para los microservicios abre nuevas posibilidades de poder aprovechar potenciales recursos proporcionados por algunos proveedores de servicios en la nube (en adelante referenciados como proveedores *cloud*) y que pueden ser interesantes y que hasta ahora no eran viables. Así lo manifiesta el autor de [19], donde relaciona las ventajas de emplear estos nuevos *cloud-based services* basados en modelos como *Infraestructure as a Service (IaaS)* o incluso *Container as a Service (CaaS)*.

Es necesario diferenciar estos nuevos servicios que han surgido a raíz del uso de contenedores y relacionarlos con las plataformas IoT propias de los que ya se había hablado y son ofertados por los mismos proveedores *cloud*.

- ***Platform 'IoT' as a Service (PaaS)***

  Se trata de un servicio en la nube administrado que permite integrar dispositivos conectados y les ofrece la posibilidad de interaccionar de manera sencilla a través de recursos propios como APIs y colas de mensajerías con otras aplicaciones en la nube y otros dispositivos.

  Es necesario volver a destacar que la utilización de estos servicios implica una elevada dependencia con el proveedor *cloud* al tener que adaptar las comunicaciones de los dispositivos a los recursos proporcionados y complica la migración de los recursos a otro proveedor debido a la dificultad que supone volver a realizar una implementación desde cero.

  Además, presenta limitaciones serias que no permiten abordar el problema adaptación que se busca desarrollar y por tanto no cumple el alcance del proyecto y limitan la propuesta de la solución.

  Otros ejemplos distintos a los que ya se han mencionados anteriormente de Amazon, Microsoft o Google son IBM con Watson IoT Platform, Cisco con IoT Cloud o Oracle con IoT Platform.

- ***Infraestructure as a Service (IaaS)***

  Se trata de un servicio que suministra infraestructura bajo demanda en forma de recursos de cómputo, almacenamiento o virtualización mediante un modelo de pago por uso. El usuario es responsable de la administración de las maquinas virtuales, el sistema operativo que se instale dentro de ellas y de cualquier tipo de aplicación o uso que se le dé.

  El proveedor proporciona un acceso a la administración de la infraestructura a través de un panel de control o una API donde podrá modificar ciertos aspectos relacionados con la red de la infraestructura, los tipos de servidores disponibles o el tipo de virtualización elegida.

  Por tanto, suele ser habitual en la mayoria de casos el poder instalar cualquier necesidad en las máquinas virtuales y por este motivo nada impediría al usuario el

poder configurar una herramienta de despliegue de aplicaciones mediante contenedores.

Algunos ejemplos son Amazon con AWS Elastic Cloud Computing (EC2), Microsoft por otra parte con Azure Virtual Machines o Google con Google Cloud Platform (GCP).

- ***Containers as a Service (CaaS)***

  Se trata de un servicio que permite implementar, administrar y escalar aplicaciones a través de una herramienta de orquestación que permite la gestión de arquitecturas de contenedores complejas en *clusters*. La principal razón de utilizar estos servicios con los orquestadores ya integrados es por la simplicidad que supone realizar una configuración mínima y el despliegue de los contenedores en infraestructura altamente disponible tal y como sostienen especialistas en diferentes publicaciones [20].

  Este tipo de servicios aprovecha la naturaleza de los contenedores que permite describir una imagen de microservicio una única vez y luego poder desplegarla tantas veces como se requiera para escalar de forma muy rápida.

  A diferencia de un IaaS, se proporciona de un servicio administrado para la ejecución de contenedores lo que permite que la parte de la infraestructura física y máquinas virtuales sea transparente para el usuario y los propios servicios.

  Algunos ejemplos de proveedores *cloud* con estos servicios son Amazon con Elastic Container Service (ECS) o Microsoft con su servicio Azure Container Service (ACS) (ya retirado).

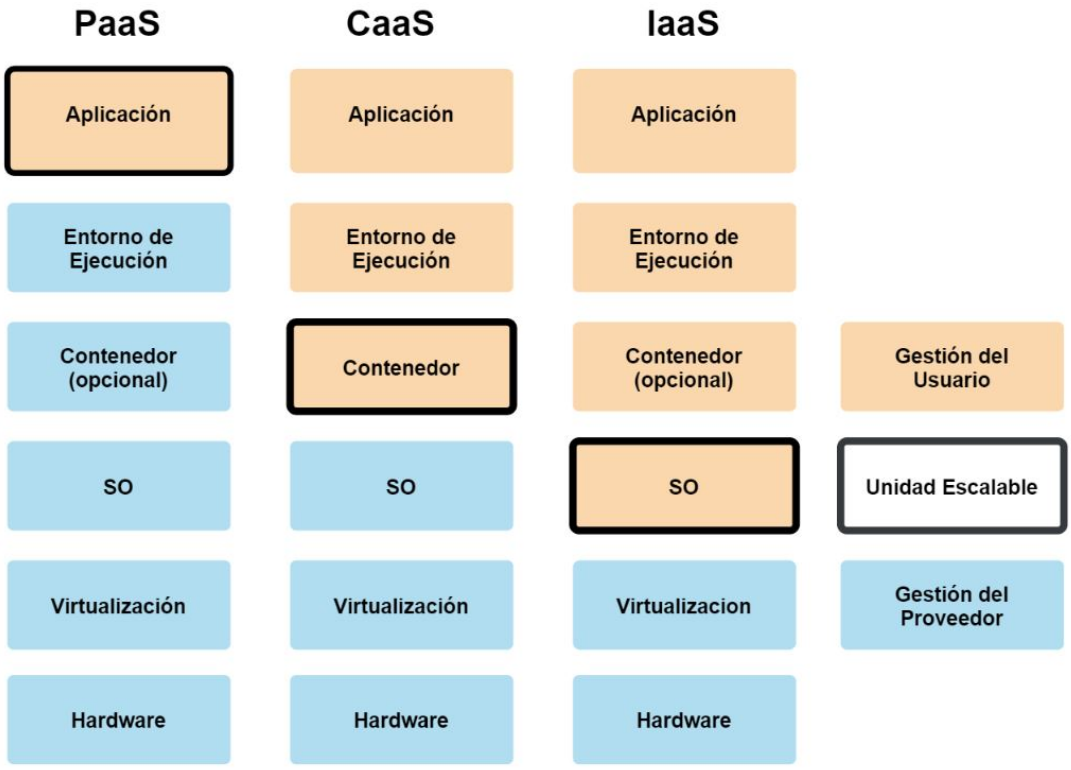


Figura 2.4: Modelos de servicios en la nube: PaaS, IaaS y CaaS

2.5 Herramientas de orquestación de contenedores

Estos Containers as a Service (CaaS) que ofrecen los proveedores tienen un precedente que es el que ha establecido la definición de la tecnología central y su implementación final como uno de los modelos claves para la nube del futuro.

Fue en el año 2014 cuando Google presentó Kubernetes [22], la primera implementación de una herramienta de orquestación de contenedores que respondía a la necesidad por parte de los desarrolladores de abstraer a los contenedores de aplicaciones de los detalles del sistema e infraestructura donde se estaban ejecutando.

En las primeras implementaciones de Kubernetes se permitía usar contenedores de tipo Docker para que utilizarán unos recursos a demanda, pero de una forma simplificada donde el desarrollador solo tenía que especificar detalles de infraestructura como el número de núcleos y memoria a utilizar según sostienen algunas publicaciones[23].

El concepto ha evolucionado los últimos años hasta permitir que la herramienta sea capaz de pasar de un modelo de ejecución donde un único host ejecuta todos los contenedores, a otro modelo donde un *cluster* de nodos distribuidos y orquestados pueden ser ejecutarlos en múltiples zonas de disponibilidad [22].

Hoy en día, se ha extendido tanto el uso Kubernetes entre los desarrolladores y las empresas que se ha convertido en una herramienta frecuentemente utilizada en algunos sectores y aunque muchos de los principales proveedores *cloud* han desarrollado sus propias alternativas e implementaciones, han acabado por integrarla como un servicio más dentro de su propio catálogo de herramientas. Un ejemplo es el de Amazon AWS con Elastic Kubernetes Service (EKS), el de Microsoft que sustituyó su servicio propio por el de Azure Kubernetes Service (AKS) y otros como Google que ya lo ofrecían desde un principio con Google Kubernetes Engine (GKE).

Se puede ver una representación de un *cluster* de contenedores para un grupo de hosts en la figura 2.5. Uno de los enfoques que caracteriza a Kubernetes del resto, es su forma de agrupar un grupo de contenedores dentro un *'pod'* e interconectar cada uno de ellos con una dirección IP única que es accesible desde cualquier otro *'pod'* aunque no estén ubicados en la misma infraestructura.

A raíz de la popularización de estas nuevas herramientas, la empresa que implementó la tecnología base de Docker también inició el desarrollo de una solución propia para integrarla dentro de su propio ecosistema y ofrecer una alternativa llamada Docker Swarm. Reconvertida y presentada como una funcionalidad integrada dentro de Docker Engine, el modo swarm es una herramienta que a diferencia del resto se centra en proporcionar un modelo *Container as a Service* (CaaS) que se pueda integrar en el propio centro de datos o nube privada de las empresa o incluso en despliegues de infraestructura de nube hibrida.

Ya bien sea con Kubernetes o con cualquier otro servicio orquestador de contendores que proporcionen los principales proveedores *cloud*, este tipo de soluciones deben de respetar una arquitectura y configuración común sobre la infraestructura donde despliegan

los contenedores tal y como sostiene el autor del artículo [19]. Algunos de los aspectos claves en cuanto a los *clusters* de contenedores y la interacción entre los diferentes elementos que lo componen son:

- Un *cluster* de contenedores está compuesto por un conjunto de nodos de cómputo. Estos nodos habitualmente son servidores virtualizados a través de instancias y que se ejecutan bajo un hipervisor.

- Cada nodo puede llegar a contener más de un contenedor.

- Un contenedor puede adoptar el rol de ofrecer un servicio único o el de ser un componente funcional del *cluster*.

- La aplicación que se ejecuta en el *cluster* se integra como una unidad compuesta por todos los contenedores.

- Los contenedores pueden montar volúmenes en aquellos servicios que requieran persistencia de datos.

- Comunicación entre los contenedores.
  Durante el diseño de la arquitectura y la implementación de los servicio se recomienda que solo uno de los nodos tenga comunicación con el exterior del *cluster* y que sea el que distribuya el resto de las comunicaciones mediante una red privada al resto.

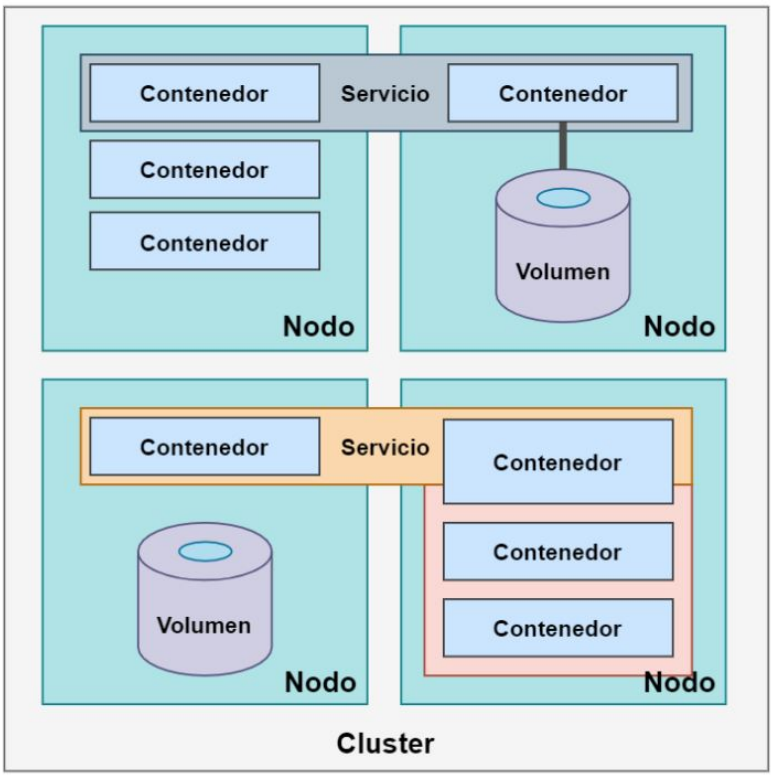


Figura 2.5: Arquitectura de un cluster de contenedores

2.6 Computación autónoma y sistemas auto adaptables

La computación autónoma se define como aquella área de investigación que estudia reducir la complejidad de los sistemas relacionando el uso de la tecnología para manejar a la propia tecnología. Es decir, la implementación de nuevos métodos, técnicas y modelos de investigación que permitan que un sistema complejo se gestione de forma autónoma a sí mismo o de forma parcialmente semiautónoma con la mínima interacción humana posible.

El objetivo de conseguir esta mínima interacción humana en sistemas que pueden llegar a tener capacidades autónomas de autogestión, es la de permitir que las empresas puedan delegar algunas funciones concretas que hasta ahora venían desempeñando sus recursos humanos (profesionales o administradores IT) para que puedan centrarse en otras tareas que aporten más valor al negocio [25].

Una primera reflexión puede encontrarse en [26], donde se manifiesta el hecho de que estas capacidades de autogestión de los sistemas informáticos se consiguen gracias a la ejecución de unas tareas que los propios profesionales IT han automatizado con tecnología de acuerdo con unas políticas. Un ejemplo de política adaptable determinaría que el tipo de decisión y acción a realizar se debería de realizar de forma autónoma en base a un aprendizaje de la información de su entorno.

El documento sostiene diferentes aproximaciones relacionadas a la autogestión destacando y relacionándolas principalmente con políticas adaptables donde la toma de acciones se basa principalmente en el análisis de diferentes situaciones y se implementan mediante otras tecnologías como diferentes bucles de control enmarcados dentro de la teoría de control [27].

Dos de las aproximación más significantes que se pueden encontrar en [26] y que mejor se relacionan con las tecnologías analizadas son:

- **Auto configuración**

  Aproximación que define la capacidad de un sistema de poder adaptarse a entornos cambiantes.

  Los componentes del sistema se adaptan dinámicamente, es decir, durante su ejecución siendo capaces de reaccionar a los cambios en el entorno utilizando las políticas establecidas.

  Dichos cambios pueden pasar por desplegar nuevos componentes, eliminar componentes ya existentes o realizar cambios en las características y arquitectura del propio sistema.

- **Auto optimización**

  Aproximación que define la capacidad de monitorización autónoma y el ajuste de los recursos de forma automática con el objetivo de satisfacer una necesidad.

El sistema tiene la capacidad de poder reasignar recursos a sus componentes como respuesta a una necesidad de forma de carga variable de trabajo y con el objetivo de poder garantizar que el servicio y funcionalidad que ofrece se pueda completar sin problemas.

Disponer de esta característica en el propio sistema, permite que no se sobredimensionen componentes que en realidad no son necesarios al disponer de un conocimiento lógico de la arquitectura de los servicios.

Analizando estos conceptos de computación autónoma se puede ver a simple vista la complejidad que supone su aplicación y los años que lleva investigándose, atendiendo a los diferentes años de publicación de los diversos artículos técnicos y documentos referenciados. La aplicación de algunas de estas aproximaciones tiene un gran potencial y pueden realizar una gran aportación a las tecnologías que se han ido desarrollando a lo largo del estado del arte como la Internet de las Cosas y la Computación en la nube.

Las soluciones que toman en cuenta estas consideraciones y consiguen aplicarlas durante el desarrollo de sistemas y aplicaciones, se denominan auto adaptables. De acuerdo con el autor de [28], define estas soluciones como aquellos sistemas que requieren de mucha información por parte de su operativa en tiempo real y que tienen la capacidad de entender lo que ocurre a su alrededor. Además, va más allá afirmando que son capaces de aprender de las situaciones por las cuales se ven afectados, pudiendo en algún momento dado generar una respuesta a estos cambios del entorno de forma autónoma.

Conseguir una adaptabilidad automática para sistemas con un funcionamiento en tiempo real es muy difícil, ya que hay que tener en cuenta que las reglas y los diferentes planes de adaptación se deben de elaborar durante la monitorización continua y propia que el sistema realiza sobre sí mismo.

Una opción más factible y que si es posible aplicar de forma sencilla es una auto adaptabilidad estática donde los mecanismos de adaptación y las reglas de actuación son definidas explícitamente por el diseñador. El sistema solo debe de preocuparse de escoger el plan más adecuado a su situación actual y aplicar las reglas que más le beneficien de acuerdo con las que tiene disponibles.

Un modelo propuesto por la empresa IBM que se ha referenciado anteriormente en [26], detalla y estructura en profundidad la composición básica que tienen los sistemas auto adaptables:

- **Autonomic Manager**

  Se trata del componente que mediante la utilización de bucles de control es capaz de "pensar" de forma autónoma y automatizar la gestión de los elementos del sistema.

  El bucle de control que está gobernado por una política definida puede incluir funciones de monitorización a otros componentes, análisis del entorno, planificación de adaptación y/o ejecución de tareas.

  Estos componentes se ponen en contacto con los *touchpoints* 'puntos de contacto' cuando ya han tomado una decisión y retransmiten la acción a ejecutar.

Algunos ejemplos de tareas que puede decidir realizar el bucle de control son:

- **Auto configuración**
  Cuando un dispositivo entra en el sistema al recupera la conectividad y es necesario desplegar su servicio y dependencias asociadas.

- **Auto reparación**
  Cuando un dispositivo sale del sistema y es necesario que otro dispositivo asuma la funcionalidad que realizaba.

- **Auto optimización**
  Cuando es necesario ajustar la carga de trabajo y proporcionar más recursos.

- **Auto protección**
  Cuando es necesario retirar un dispositivo del sistema al detectar que tiene un comportamiento errático.

- ***Touchpoints***

  Se trata del componente que necesita el bucle de control para poder monitorizar y ejecutar las acciones sobre los elementos del sistema.

  El *touchpoint* implementa una interfaz de gestión estándar que permite manejar y controlar a los elementos del sistema. Compuesto por sensores que recogen información y efectores o actuadores que ejecutan operaciones a los componentes de la solución a través de sus interfaces nativas.

  Algunas operaciones que se le pueden solicitar son:

  - **Desplegar un servicio**
  - **Retirar un servicio**
  - **Reconfigurar un servicio**
  - **Redimensionar un recurso**

- ***Managed resource***

  Un recurso gestionado puede tratarse de un componente físico, un servicio lógico asociado a un dispositivo físico o un conjunto de servicios de un sistema mucho más grande.

  Se trata del componente de los sistemas auto adaptables que esta administrado por los otros dos componentes que gestionan su adaptación.

La estructura definida en el artículo [26], permite definir un estándar de facto en el desarrollo de estos sistemas al conceptualizar una separación lógica de la solución en dos partes. Una primera parte compuesta por el sistema base o solución de la que no se requiere que se modifique su implementación y que se encuentra administrada, y otra segunda parte que implementa una capa de adaptación compuesta por el bucle de control y los sensores y efectores.

2.7 Infraestructura como código

El alcance del proyecto define desde el comienzo la utilización de servicios de infraestructura en la nube para el despliegue de la solución de forma que se pueda aprovechar algunos servicios relacionados con el rendimiento, la elasticidad y la escalabilidad. Integrar la solución de microservicios dentro de contenedores facilita realizar un despliegue en este tipo de infraestructuras.

A consecuencia de esto, el desarrollo del componente "*touchpoint*" visto en la estructura de un sistema auto adaptable, se ve afectado al necesitar controlar la arquitectura lógica de la solución y su necesidad de realizar operaciones que requieran desplegar o retirar un servicio, reconfigurar los parámetros de comunicación entre servicios o incluso parametrizarlos.

Las herramientas Infrastructure as Code (IaC) permiten abordar este problema y realizar una gestión sobre la infraestructura al describirla en una especie de modelo detallado como si fuera una receta. El objetivo es proporcionar de un instrumento a los desarrolladores que les permita administrar, monitorizar y aprovisionar recursos de forma automática en lugar de realizar una configuración manual donde se pueden producir errores humanos. Además, de la ventaja de poder desplegar la misma receta varias veces y conseguir que el entorno siempre sea el mismo.

La principal ventaja del uso de estas herramientas es que permiten realizar un aprovisionamiento y despliegue en infraestructura de proveedores *cloud*, permitiendo que a través de recetas o *templates* se describa la configuración y la topología de recursos tales como máquinas virtuales, almacenamiento o recursos de red entre otros.

Un ejemplo es el de Cloudformation, una herramienta desarrollada por Amazon para operar en su *cloud* público Amazon Web Services. Las recetas de infraestructura están descritas en un formato de descripción JSON o YAML y permiten crean unos recursos virtuales que son agrupados en *stacks* 'pilas' que facilitan de una forma muy efectiva controlar la dependencia entre los recursos, incluso pudiendo gestionar su conjunto a través de operaciones tipo Create, Read, Update and Delete (CRUD).

Otra herramienta ampliamente extendida es Terraform desarrollado por Hashicorp, que despunta gracias a su flexibilidad y posibilidad de poder operar en múltiples proveedores *cloud* como Amazon AWS, Microsoft Azure o Google GCP. Las recetas están descritas en su propio lenguaje HCL y destaca su gran modularidad y administración eficiente del estado de los recursos.

2.8 Propuesta

La propuesta trata de aplicar todo el análisis de investigación realizado a lo largo del estado del arte con el objetivo de aclarar y justificar cuál es el espacio de conocimiento que va a cubrirse y que conjunto de tecnologías o herramientas van a facilitar el diseño de una solución con la que abordar el problema.

En más de una ocasión se ha destacado que las herramientas y plataformas que ofrecen los principales proveedores *cloud* para gestionar dispositivos IoT son insuficiente para resolver cualquier problema de adaptación y reconfiguración debido a esa falta de control que se tiene sobre la infraestructura donde se están ejecutando las copias virtuales.

Por tanto, debe plantearse el desarrollo de una solución genérica que sea capaz de extenderse sin dificultad para adaptarse a cualquier situación y que pueda ser desplegada en cualquier tipo de infraestructura o plataforma. Utilizar una arquitectura basada en microservicios además de sus múltiples ventajas ya descritas en este mismo capítulo, permite poder hacer una relación directa dispositivo-servicio que facilita la implementación de estas soluciones de carácter distribuido. Al mismo tiempo, encapsular la solución de servicios desarrollados dentro de contendores facilita que puedan ser ejecutados y desplegados de forma rápida y segura en cualquier entorno de cómputo que disponga de las herramientas para ejecutarlos.

Los modelos de servicio CaaS que ofrecen los principales proveedores *cloud* permiten poder ejecutar esta solución de contenedores a través de herramientas de orquestación en una infraestructura elástica y distribuida en la nube simplificando el despliegue donde el desarrollador solo tiene que especificar detalles mínimos de infraestructura como el número de núcleos o memoria a utilizar.

El uso de contenedores junto con una adecuada implementación de las comunicaciones entre microservicios permite poder integrar cualquier tipo de tecnología en la solución convirtiéndose en un modelo fácilmente extensible. Por esta razón, desarrollar capas de adaptación por encima de la solución es más sencillo de lo que se puede pensar, donde únicamente hay que preocuparse por desarrollar componentes de sensores, efectores o bucles de control que puedan comunicarse y entenderse con el resto.

Los sensores y efectores de la adaptación necesitan poder realizar operaciones sobre la infraestructura y la arquitectura de la aplicación en tiempo de ejecución, para automatizar procesos de gestión de recursos que están desplegados en la nube. La utilización de herramientas de infraestructura como código facilitan describir, modificar y aprovisionar de manera programática estos recursos de manera automática y cumplir con las necesidades funcionales que requieren los componentes de adaptación.

El resultado pretende relacionar una combinación de estas tecnologías existentes y realizar una prueba de concepto para ver hasta que punto es posible preparar que una infraestructura de nube pública pueda ser adaptada y gestionada por tecnologías de computación autónoma.

CAPÍTULO 3
Análisis del problema

3.1 Arquitectura de referencia para el desarrollo de sistemas auto adaptables

Se toma como punto de partida para abordar el problema las descripción planteada por IBM en el documento [26] que propone una arquitectura de referencia para ayudar al desarrollo de soluciones que apliquen conceptos de computación autónoma en sistemas informáticos. La importancia de esta arquitectura radica en un enfoque práctico que centra el desarrollo de nuevos sistemas que son capaces de realizar tareas de autogestión utilizando modelos de bucles de control alimentados por el conocimiento que obtienen desde su entorno.

Disponer de esta referencia facilita, en gran medida, el objetivo perseguido en el proyecto de analizar y abordar la problemática del desarrollo de sistemas compuestos por dispositivos IoT, que necesitan adaptarse y reconfigurarse para responder ante factores externos de terceros aprovechando toda la información que tienen disponible.

Durante la primera fase se ha realizado un primer análisis del documento que describía la necesidad de construir dos grandes bloques arquitectónicos adicionales para construir esas capacidad autónomas sobre un tercer bloque que es el sistema que es necesario que se adapte.

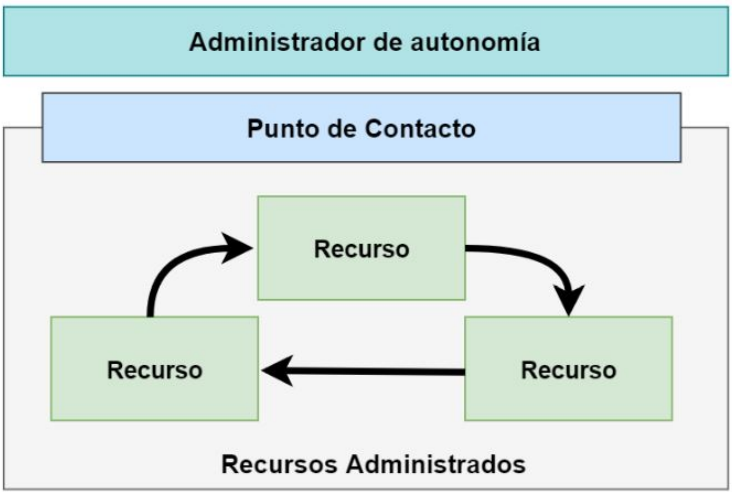


Figura 3.1: Modelo de sistema auto adaptable

En la figura 3.1 se puede ver como una capa inferior representa a los componentes del sistema como *managed resources* 'recursos administrados' y conforman la solución principal. Otra capa intermedia incorpora interfaces de gestión para acceder y controlar a los recursos administrados a través de los *touchpoints* 'puntos de contacto'. Finalmente, una capa superior es la que integra unos bucles de control que proporcionan la capacidad de decisión a través de un administrador de autonomía (Autonomic Manager).

3.2 Autonomic Manager: El bucle de control MAPE-K

Un administrador de autonomía es un componente externo que permite automatizar algunas funciones de gestión de un sistema manejado. Esta automatización viene dada gracias a la implementación de un bucle de control que necesita ser alimentado con la máxima información posible del sistema para poder analizarla y determinar si es necesario cambiar algo y planificar una serie de respuestas. Este bucle de control es el que proporciona la inferencia al sistema.

Como se muestra en la figura 3.2, el bucle de control denominado MAPE-K esta compuesto de cuatro partes que son las que trabajan y colaboran entre sí intercambiando datos con un conocimiento común para poder dar una funcionalidad de control.

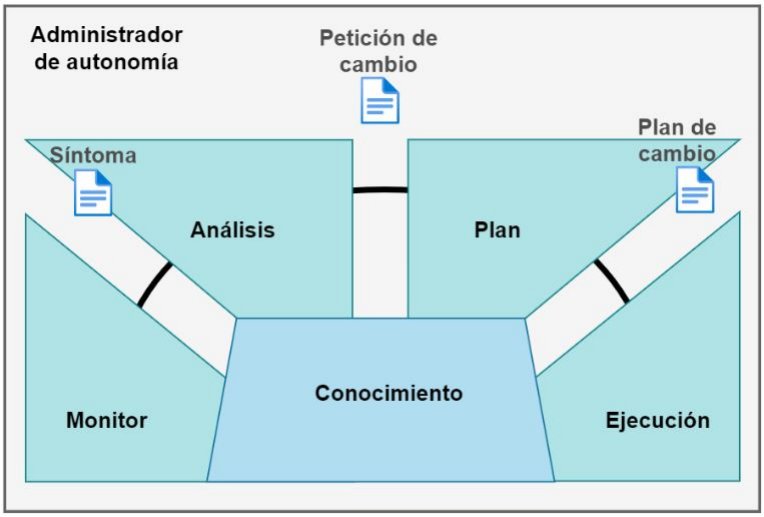


Figura 3.2: Arquitectura auto adaptativa: Modelo adminsitrador de autonomía

- **Monitor**

  El monitor es la función que a través del *touchpoint* recoge información sobre la configuración o estado de los recursos administrados. Esta función también recopila una gran cantidad de datos relacionados con la capacidad y el rendimiento de los procesos y su objetivo es realizar un primer diagnóstico o filtro superficial para determinar si hay algún síntoma por parte del sistema que se necesita ser analizado con más detalle y profundidad.

- **Análisis (Analyze)**

  El análisis es la función que proporciona las herramientas para determinar la presencia de situaciones que precisan una respuesta porque una de las políticas definidas no se esta cumpliendo. Este tipo de técnicas suelen llevar una complejidad

asociada que es muy elevada debido a que necesitar determinar en tiempo real si es necesario realizar algún cambio en el momento o en el futuro. La respuesta puede requerir la necesidad de realizar un cambio de comportamiento describiendo los aspectos del sistema que son necesarios adaptar.

- **Plan**

  La función de planificación recibe un requerimiento de cambio de uno o varios aspecto del sistema y es el encargado de determinar cual es el mejor procedimiento para realizar esas adaptaciones. Un requerimiento de cambio puede adoptar muchas formas, incluso la de volver a reconfigurar y desplegar la solución desde cero. El resultado es la generación de un plan de cambio que se debe de ejecutar.

- **Ejecución (Execute)**

  La ejecución es la función que recibe como entrada un plan de cambio y determina cuales son las acciones concretas, necesarias y ordenadas para poder llevar a cabo punto por punto el cambio. Se encarga de que las acciones resultantes sean transmitidas una a una al *touchpoint* para que las realice y esperará la respuesta de los resultados antes de volver a procesar la siguiente acción.

- **Conocimiento (Knowledge)**

  Se trata de la fuente de conocimiento que funciona como un registro donde se almacenan las conclusiones de cada una de las fases del bucle de control para que estén accesibles para el resto. Alguna de estas conclusiones son síntomas, políticas, solicitudes de cambio y planes de cambio.

3.3 *Touchpoint*: Sensores y efectores

Un *touchpoint* es otro de los componentes externos que es necesario dentro de una solución auto adaptable y que permite exponer el estado de un sistema gestionado al administrador de autonomía. Cuando el administrador de autonomía ha elaborado una respuesta a través de su bucle de control, comunica al punto de contacto cuales son las operaciones necesarias que necesita ejecutar para adaptarse.

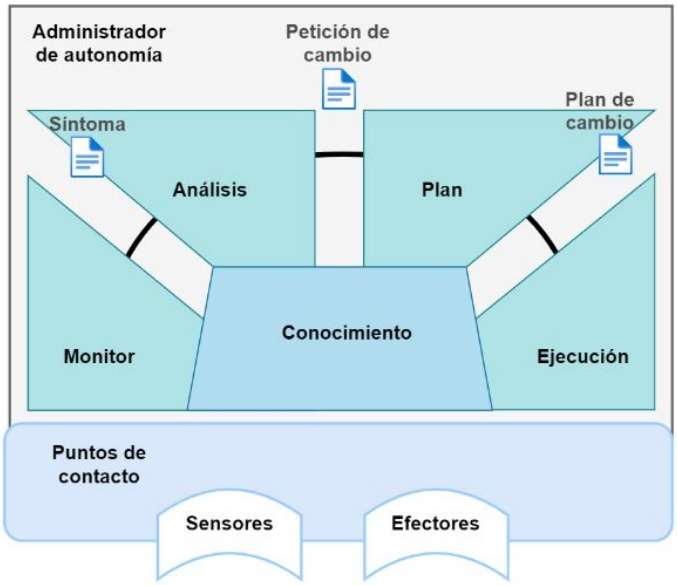


Figura 3.3: Arquitectura auto adaptativa: Modelo punto de contacto

Tal como se puede ver en la figurar 3.3, una implementación de una interfaz de gestión básica es la que permite al *touchpoint* recopilar información a través de sensores y ejecutar operaciones a través de los efectores sobre el sistema administrado. Estos sensores y efectores están directamente relacionados con las diferentes interfaces nativas y tecnologías de los recursos administrados y adaptan para cada uno de ellos su comunicación para obtener el resultado que se espera.

El objetivo de tener esta capa e interfaz de gestión intermedia es que el recurso administrado permanezca inalterada su implementación y sea el *touchpoint* el que se adapte a las comunicaciones e interfaces de los propios recursos que está administrando.

Un sensor se caracteriza por contar con las siguientes características inherentes:

- Se expone un conjunto de detalles y características de un recursos administrado a través de una operación de consulta estándar como puede ser un *GET*.

- Una serie de alertas y eventos previamente configurados notifican modificaciones detectadas en el estado cuando se producen cambios de alguna o varias características del recurso adminsitrado.

- La implementación de la operación *GET* para realizar consultas del estado de un recurso administrado pueden cambiar dependiendo de la tecnología utilizada por los otros recursos.

De la misma forma, se considera que un efector se caracteriza por contar con las siguientes características:

- Se expone un conjunto de operaciones *SET* que permitan alterar las características de un recurso administrado.

- Cuando el administrador de autonomía decide realizar un cambio en la solución hace uso de estas operaciones.

- La implementación de las operaciones *SET* para modificar el estado de un recurso administrado puede cambiar dependiendo de la tecnología utilizada por otros recursos.

Estos dos componentes están interrelacionados de forma estrecha ya que un cambio efectuado en el sistema a través de un efector tiene que ser detectado por los eventos y alertas configuradas en el sensor.

3.4 Recurso administrado: *adaptive-ready* y auto adaptativo

Un recurso gestionado puede tratarse de un componente físico, un servicio lógico asociado a un dispositivo físico o un conjunto de servicios de un sistema mucho más grande como es el caso que nos aplica en este proyecto. Se trata del componente de los sistemas auto adaptables que no necesita ser modificado y mantener su implementación original ya que la autonomía es proporcionada por las otras dos capas y sus componentes como el bucle de control 'MAPE-K' o los sensores y efectores.

Sin embargo, aunque no se requiera la modificación del sistema administrado si que es necesario que como mínimo sus diferentes recursos dispongan de alguna tecnología nativa con la que los elementos del *touchpoint* (sensores y efectores) puedan interactuar y consultar su estado o características del sistema.

Es importante destacar dos conceptos con respecto a los recursos administrados:

- **Recurso *adaptive-ready***

  Este tipo de recurso implementa una capa superior de adaptación de sensores y efectores que proporciona una interfaz de gestión sobre el sistema.

- **Recurso auto-adaptativo**

  De la misma forma, este recurso también se ha implementado una capa de sensores y efectores pero con un bucle de control 'MAPE-K' que añade la posibilidad de tomar decisiones de forma autónoma.

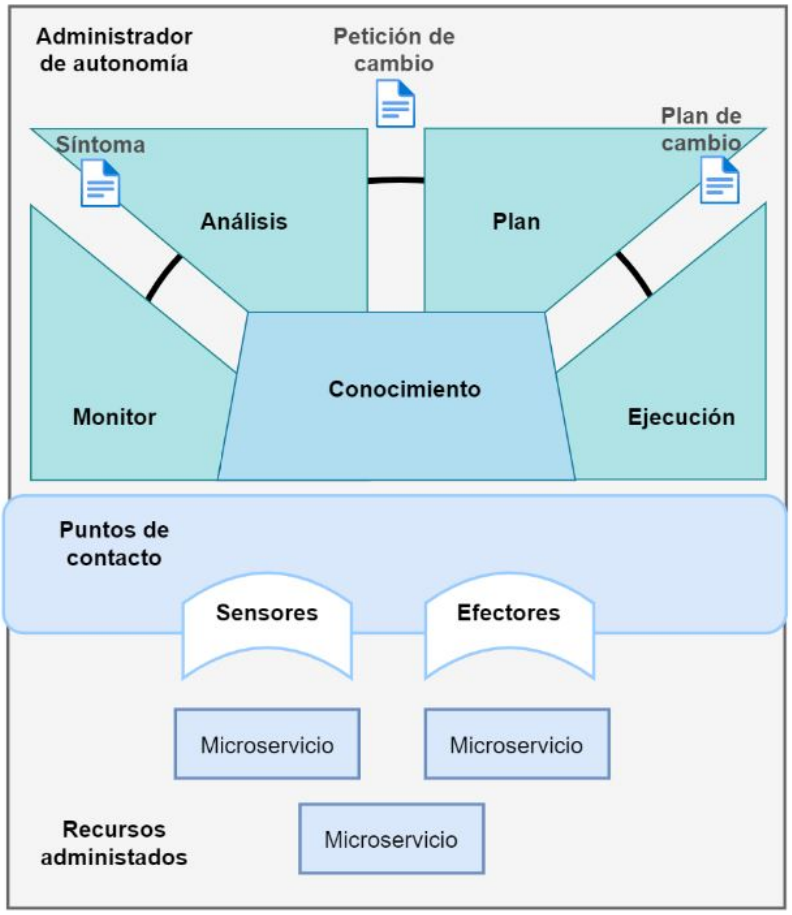


Figura 3.4: Arquitectura auto adaptativa: Modelo recurso administrado

3.5 Relación de la arquitectura con la solución de microservicios

A través de la siguiente tabla 3.1 se representa visualmente de forma sintetizada los componentes necesarios que se han detectado al estudiar la arquitectura y relacionarla con la solución propuesta.

La primera columna denota la referencia de los modelos de la arquitectura, en la segunda los recursos a desarrollar y en la última, los resultados que se esperan obtener a medida que avanza el desarrollo.

Arquitectura de referencia aplicada a la solución propuesta		
Modelo	Recurso	Aplicación
• Autonomic Manager	Bucle de control MAPE-K	Bucle MAPE-K + Sensores y efectores + microservicios = Solución de microservicios auto-adaptativos
• *Touchpoint*	Sensor y Efector	Sensores y efectores + microservicios = Solución de microservicios *adaptive-ready*
• *Managed resource*	Microservicio	Solución de microservicios

Tabla 3.1: Arquitectura de referencia de soluciones auto adaptativas aplicada a la solución propuesta

Con solo la implementación de una capa de sensores y efectores a la solución de microservicios ya se consigue que el sistema sea ***adaptive-ready*** y por tanto la solución esté preparada para recopilar información y aplicar operaciones sobre los microservicios a través de una interfaz de gestión.

Cuando se desarrolla el bucle de control de tipo MAPE-K, se extienden estas funcionalidades utilizando técnicas y métodos propios de la computación autónoma para conseguir que la solución sea **auto adaptable**.

3.6 Relación de la arquitectura con la infraestructura en la nube

Desarrollar una solución de microservicios aplicando tecnología de contenedores permite entre otras cosas poder desplegar el sistema en una infraestructura elástica en la nube de forma sencilla sin depender de ellos al completo. Esto es posible gracias a las herramientas de orquestación de contenedores que ofrecen algunos proveedores *cloud* a través de sus servicios Cointainer as a Service (CaaS) que permiten mantener una buena calidad de servicio y abstraer la configuración de la infraestructura a la propia solución.

Sin embargo, a pesar de todas las ventajas que ofrecen aun no garantizan el poder hacer una reconfiguración durante su ejecución (en caliente) de la arquitectura de la solución, que es el problema que se trata de investigar a lo largo del proyecto. Es necesario analizar hasta que punto estas herramientas y proveedores *cloud* permiten poder desarrollar esta funcionalidad externamente y poder preparar una infraestructura virtual para que sea *adaptive-ready* aplicando los conceptos que se han visto de computación autónoma.

Con el objetivo de limitar el alcance del proyecto y poder centrarse en la principal línea de investigación junto con el cumplimiento de los objetivos previamente definidos, será necesario realizar una selección de servicio y proveedor *cloud* para desplegar la solución y la adaptabilidad. Este es uno de los motivos por los es necesario descartar herramientas que no son ofrecidas por proveedores *cloud* como algunas implementaciones de Kubernetes (*no cloud-based services*) o Docker Swarm ya que requieren de una configuración previa e infraestructura dedicada disponible.

También, hay que tener presente que los efectores necesitan interactuar y controlar los recursos virtuales desplegados en la nube para poder realizar cambios que pueden requerir en algunos casos que se realice una reconfiguración en la arquitectura lógica de la solución. Es necesario que el proveedor seleccionado proporcione una perfecta integración con herramientas de infraestructura como código y API de desarrollo para los principales lenguajes de programación que sean capaces de realizar operaciones que impliquen desplegar o retirar un servicio, reconfigurar los parámetros de comunicación entre varios servicios o redimensionar sus recursos.

3.7 Computación *Serverless*

Las plataformas *serverless* computing (computación sin servidor) son aquellas que utilizan las herramientas de orquestación en la nube para poder ejecutar contenedores sin tener que preocuparse de la compleja gestión de una infraestructura compuesta por un *cluster* de instancias y máquinas virtuales.

Esta tecnología consigue que la solución aprovisione los recursos virtuales de forma automática, eliminando la necesidad de especificar uno de los diferentes tipos de servidores a utilizar y a definir reglas de escalado automático necesarias para la optimización de los recursos.

No obstante, sí que es necesario definir una configuración e infraestructura mínima de red mediante la cual los diferentes contenedores pueden comunicarse entre sí. Esto es posible gracias a la creación de de recursos de red virtuales llamados Virtual Private Cloud (VPC) de los proveedor *cloud*, que garantizan que la utilización de la infraestructura virtuales sea bajo una red independiente y totalmente configurable por el usuario.

Hay que tomar en consideración a la hora del desarrollo de la solución que este tipo de plataformas está orientado a la ejecución de contenedores con servicios sin estado (stateless services) y no soportan el uso de volúmenes persistentes.

Si es necesario mantener el estado hay que delegar esas funcionalidad de persistencia a servicios externos de almacenamiento y bases de datos que normalmente también proporcionan los propios proveedores *cloud*. La configuración de red debe considerar el acceso y comunicación con este tipo de recursos externos.

Tras el estudio de las diferentes alternativas existentes en el mercado, se ha desarrollado una comparativa (tabla 3.2) que destaca las principales diferencias y funcionalidades ofrecidas por estas plataformas y relacionándolas con los servicios de orquestación de contenedores con el objetivo de poder determinar cuál es la solución que mejor encaja con el proyecto.

Funcionalidad	AWS Fargate	Azure Container Instances	Google Cloud Run
Herramienta de orquestación	AWS ECS o AWS EKS	Azure AKS	Google GKE
Despliegue	Imagen contenedor o Kubernetes pod	Imagen Contendor	Imagen contendor
Aislamiento	Máquina Virtual	Máquina Virtual	gVisor
Multicontenedor	Si	Si	No
Infraestructura como código	Cloudformation Terraform	Terraform	Terraform
Conectividad de red	Amazon VPC *Service Discovery*	Azure Virtual Networks	Google VPC

Tabla 3.2: Comparación de los principales servicios de orquestación de contenedores y computación *serverless*

Se puede ver a simple vista que todas las plataformas *serverless* soportan una orquestación de contenedores basándose en un modelo de implementación propia de Kubernetes y por este motivo casi todas las características y funcionales ofrecidas son iguales en cuanto a despliegue, aislamiento o conectividad de red.

A diferencia del resto, AWS Fargate también proporciona compatibilidad con su propio servicio de orquestación de contenedores llamado Amazon Elastic Container Service

(ECS).

Considerar la utilización de ECS para el diseño de la solución puede ser interesante ya que permite conseguir una mejor integración con el resto de los servicios del proveedor *cloud* y proporciona una simplicidad en la gestión de la herramienta que las implementaciones propias de Kubernetes no consiguen.

Adicionalmente, un factor que también contribuye en su elección podría ser la disposición de una mejor tarificación en el modelo de pago por uso que EKS y para el desarrollo del proyecto es preferible seleccionar una herramienta que permita despreocuparse de su configuración básica y permita poder centrarse en los objetivos de adaptabilidad.

CAPÍTULO 4
Diseño de la solución

A lo largo de las fases previas de investigación, se ha concluido que la aplicación de modelos de computación autónoma sobre soluciones basadas en arquitecturas de servicios distribuidos no es trivial y requiere de una definición exhaustiva.

El modelo referenciado y estudiado durante el capítulo de análisis del problema, define una arquitectura de referencia para el desarrollo de sistemas auto adaptables que indica que no es necesario realizar una modificación integra en los servicios funcionales de la solución para poder llegar a implementar conceptos de autonomía. No obstante, si que puede llegar a ser necesario y recomendable investigar hasta qué punto es posible facilitar un buen diseño de los servicios del sistema para que estén preparados para esa adaptación y puedan llegar a aplicar estos modelos de una forma óptima.

4.1 Notación para arquitecturas de microservicios

Antes de empezar con el diseño de la solución, es necesario establecer cuál va a ser la notación que se utiliza a lo largo del capítulo para poder entender los conceptos que se intentan representar mediante esquemas sobre la arquitectura de la solución. Los autores del artículo [30] definen una notación pensada para las arquitecturas de microservicios y representada mediante el conocido lenguaje de modelado Unified Modeling Language (UML).

Como ya se ha definido, las arquitecturas de microservicios están formadas por una agrupación de pequeños servicios que son independientes y proporcionan como mínimo una función específica. Estos servicios pueden encontrarse distribuidos a lo largo de varios dispositivos o nodos y se comunican entre sí mediante redes privadas con el objetivo de compartir información y proporcionar una funcionalidad en forma de solución global y común de sistema.

Una simplificación propia de la notación que se utiliza para representar de forma conceptual los microservicios y sus relaciones hasta formar una solución puede verse referenciada en la tabla 4.1.

	Notación	Descripción
Microservicio		Un microservicio se representa como un circulo de color
Interfaz: Función		Una interfaz que proporciona una función del **mismo** microservicio se representa como una línea acabada en círculo.
Interfaz: Requerimiento		Una interfaz que requiere de una función de **otro** microservicio se representa como una línea acabada en semicírculo
Enlace		Una comunicación entre diferentes microservicios se representa como una flecha continua
Solución de microservicios		Una solución de microservicios se representa como un conjunto de servicios, funciones y enlaces

Tabla 4.1: Notación para arquitecturas de microservicios

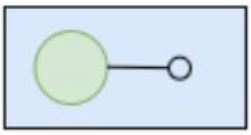
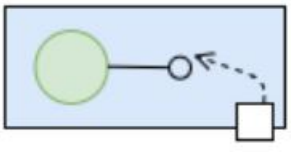
	Notación	Descripción
Contenedor		Un contenedor es un entorno de ejecución de aplicaciones y se representa como un rectángulo con fondo azul.
Contenedor: Microservicio		Un contenedor que ejecuta un microservicio se representa como un circulo dentro del rectángulo
Contenedor: Enlace		Un enlace dentro de un contenedor se representa como una línea discontinua
Contenedor: Puerto de comunicación		Una puerto de comunicación es un mecanismo que limita el acceso a los recursos disponibles dentro del contenedor. Se representa como un cuadrado pequeño en el borde del rectángulo.

Tabla 4.2: Notación para arquitecturas de contenedores

La notación también debe incluir los entornos de ejecución como los contenedores donde se ubican estos microservicios a modo de poder representar la relación entre las interfaces de red y puertos de comunicación que es necesario desarrollar. En la tabla 4.2 puede encontrarse la notación con detalle.

Por último, es necesario representar también el host o nodo que puede ser un dispositivo físico o un recurso de cómputo virtual que se encarga de la ejecución de uno o más contenedores. En la tabla 4.3 puede encontrarse la notación con detalle.

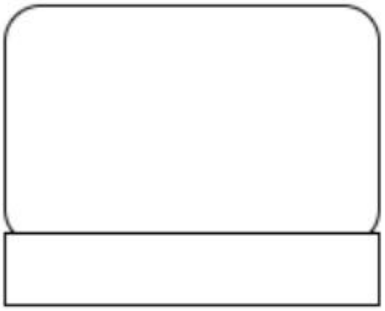
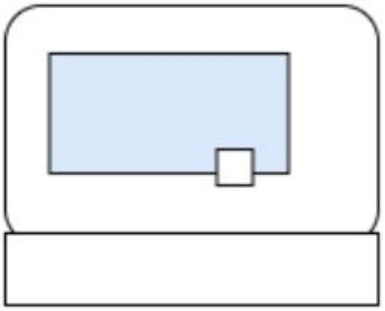
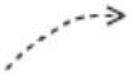
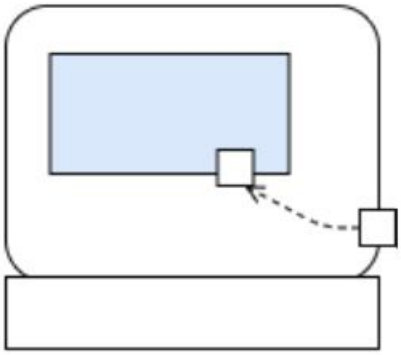
	Notación	Descripción
Host		Un host es un recurso de computo físico o virtual con la capacidad de poner en marcha contenedores. Se representa como un cuadrado con fondo blanco y un rectángulo en el pie.
Host con contenedor		Un host que ejecuta un contenedor se representa como un rectángulo azul dentro del cuadrado blanco.
Host: Enlace		Un enlace dentro de un nodo se representa como una línea discontinua
Host con puerto de comunicación		Una puerto de comunicación es un mecanismo que limita el acceso a los recursos disponibles dentro del host. Se representa como un cuadrado pequeño en el borde del cuadrado blanco.

Tabla 4.3: Notación para arquitecturas en la nube

4.2 Patrones arquitectónicos para el desarrollo de microservicios

La arquitectura de microservicios permite estructurar la implementación de una aplicación en pequeños servicios dónde cada uno de ellos proporciona una de las funcionalidades de la solución. Esta división no se puede hacer de cualquier forma, sino que debe seguir una estructura lógica para poder aprovechar sus potenciales ventajas de escalabilidad, mantenimiento y modularidad.

La utilización de patrones de diseño de arquitecturas durante el desarrollo puede ayudar a descomponer de manera óptima una solución atendiendo a los requerimientos de una aplicación ya que identificar los posibles dominios y clases no es una tarea fácil cuando se está intentando desarrollar una solución compleja. Pero no solo contribuyen durante esta división, sino que también pueden definir patrones de comunicación entre los diferentes servicios. Es en este punto donde es necesario explicar la relevancia de algunos de ellos.

4.2.1. Registrador de servicios

Normalmente los servicios dentro de una solución exponen una interfaz a raíz de la cual exponen su funcionalidad al resto de microservicios utilizando recursos API en una ubicación específica a través de una dirección de host y número de puerto.

Pero una solución de estas características implica que sus componentes están "vivos" y el número de microservicios que hay en ejecución puede llegar a ser muy variable, así como la localización de sus recursos que también cambia constantemente.

El patrón registrador de servicios (*service registry*) pretende resolver el problema de como un microservicio que requiere de la funcionalidad de otro para su operativa, pueda averiguar si está disponible y dónde poder encontrarlo.

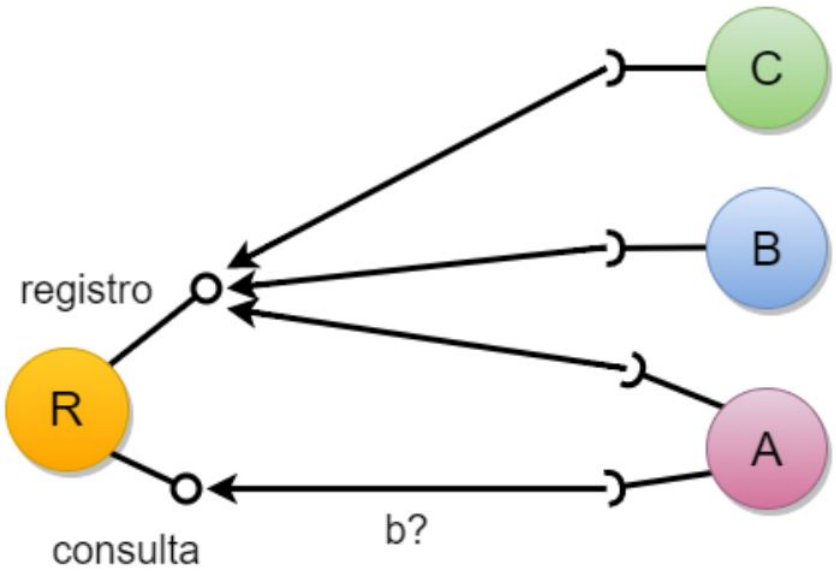


Figura 4.1: Modelo conceptual de un registrador de servicios

Como indica su nombre y se puede ver en la figura 4.1, este patrón establece que uno de los servicios adopta el rol de registrador de servicios y mantiene en una base de datos la disponibilidad y localización del resto de componentes del sistema. Su funcionamiento se define por estas cuatro siguientes reglas:

- Un servicio debe comunicar al registrador en todo momento cualquier cambio en su estado desde que se registra al entrar al sistema hasta que cancela su disponibilidad al salir del mismo.

- Cualquier servicio del sistema puede consultar en el registro la disponibilidad y localización de una instancia de servicio que proporcione la funcionalidad que está buscando.

- Se puede implementar un servicio de monitorización del tipo '*Health Check*' que compruebe periódicamente si el recurso sigue estando disponible.

- Debe ser el primer servicio que se despliegue en la solución de manera que el resto cuando inicien su funcionalidad puedan registrarse.

Un ejemplo de registrador puede ser el proporcionado por Netflix Eureka y que se puede implementar de forma sencilla a través del framework para Java de Spring Boot.

4.2.2. API Gateway

La utilización de este tipo de arquitecturas basadas en microservicios permite la utilización de diferentes tecnologías para desarrollar las funcionalidades que componen la solución. En las arquitecturas monolíticas tradicionales esto es mucho más difícil debido a que es necesario conseguir una interoperabilidad y compatibilidad con el resto de los componentes que están desarrollados y empaquetados en una única solución lo que incrementa enormemente la complejidad de su desarrollo.

Los microservicios pueden permitir la abstracción completa del concepto de cómo se produce esa funcionalidad y se centra en ofrecerla al resto, utilizando normalmente interfaces API que les permite interactuar y transmitir información en un lenguaje que todos los componentes de la solución entienden.

Sin embargo, el usuario final del sistema no tiene por qué comunicarse de forma individual con cada uno de los microservicios cuando pretende utilizar la aplicación ya que su usabilidad no sería práctica. Es más, mucho de estos microservicios no tienen porque estar públicos de cara al usuario ya que no le es de utilidad al proporcionar una funcionalidad que apoya a otros recursos.

Esta problemática se suma a la anterior, dónde los componentes de la solución cambian constantemente y las instancias de los servicios pueden cambiar sus ubicaciones en cualquier momento teniendo que volver a consultar su nueva posición cada vez que se quiera hacer uso de sus funcionalidades.

El API Gateway o puerta de enlace API, establece que un servicio sea el único punto de entrada para todos los clientes y que sea este el encargado de gestionar las solicitudes redirigiéndolas a los servicios correspondientes o que actúe como intermediario y proporcione la información él mismo.

Otros de los beneficios que aporta la implementación de este patrón son:

- Ocultar al usuario final el tipo de arquitectura con la que se implementa la solución de microservicios y, por tanto, eliminar la responsabilidad de determinar la localización las diferentes funcionalidades distribuidas.

- Reducir el número de peticiones dirigidas a los servicios al delegar esa responsabilidad a la puerta de enlace que puede optimizar las peticiones y devolver toda la información en una sola respuesta.

- Simplificar la implementación de políticas de seguridad al concentrarse todas las comunicaciones públicas en un solo nodo y aislar el resto en una red privada.

Como se puede ver en la figura 4.2, este servicio se apoya en el Registrador de Servicios para consultar la localización de los recursos. Un ejemplo de API Gateway puede ser el proporcionado por Netflix Zuul y que se puede implementar de forma sencilla a través del framework para Java de Spring Boot.

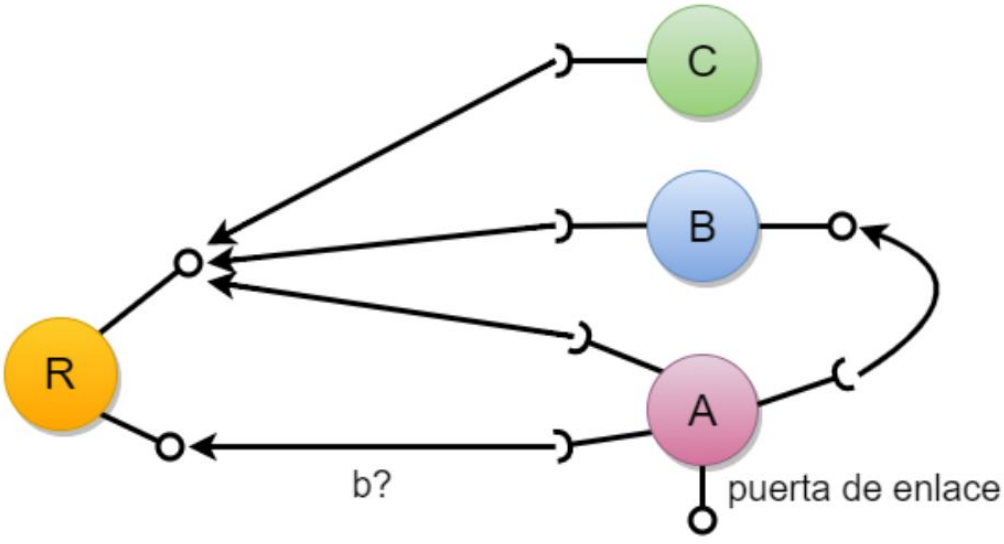


Figura 4.2: Modelo conceptual de un Api Gateway

4.3 Diseño de la solución de microservicios

El diseño de la solución aborda como base una solución de microservicios sencilla sobre la que posteriormente sea posible desarrollar la adaptación que se busca investigar. Hay que tener en cuenta que esta solución aun pretendiendo ser sencilla y genérica, debe de poder aplicarse con facilidad en entornos donde las condiciones de funcionamiento pueden llegar a ser muy cambiantes como la Internet de las Cosas y la Industria 4.0 donde los dispositivos físicos están relacionados con algunos de estos microservicios directamente.

Por este motivo, es necesario profundizar y hacer uso de patrones de comunicación que permitan gestionar una comunicación distribuida entre servicios y algunos parámetros como su identificación, localización de recursos o enrutamiento el enrutamiento de peticiones realizadas por parte de los usuarios del sistema.

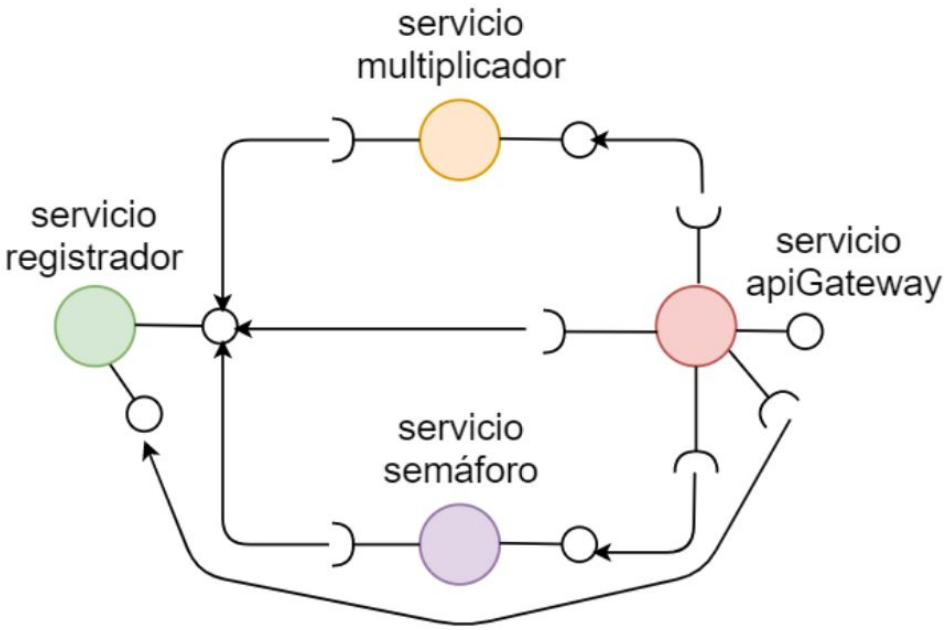


Figura 4.3: Diseño de la solución de microservicios

El sistema representado en la figura 4.3 muestra una solución de microservicios con los siguientes componentes:

- **Servicio registrador de servicios**

  Aplica el patrón arquitectónico *service registry* y es el encargado de mantener un registro de servicios indicando su disponibilidad y la localización de sus recursos.

  Dispone de dos interfaces funcionales: una es el registro de clientes y la otra es la consulta de servicios.

- **Servicio API Gateway**

  Aplica el patrón arquitectónico API Gateway y es el encargado de gestionar y redirigir las solicitudes de los usuarios a través de una puerta de enlace.

  Dispone de una interfaz funcional que proporciona una API REST con la que poder interactuar con el sistema a través de solicitudes y dos interfaces que utiliza para comunicar su estado al registrador o consultar la ubicación de un servicio.

  Por otra parte, dispone de tantas interfaces requeridas como servicios existan en la solución que proporcionen funcionalidad al usuario. En el caso expuesto se trata de doablaNumeros y semáforo.

- **Servicio multiplicador**

  Se trata de un servicio que proporciona una funcionalidad arquetípica como es la de multiplicar el número que se le proporcione como parámetro en la solicitud. Además, mantiene un contador de cliente que va incrementando en cada respuesta que proporciona. Esta funcionalidad la presenta a través de una interfaz proporcionada.

- **Servicio semáforo**

  Se trata de un servicio que está directamente relacionado con un dispositivo físico que actúa como un semáforo con sus tres luces (rojo, ámbar y verde).

  El servicio proporciona una interfaz API REST que permite interactuar y gestionar el semáforo de forma remota pudiendo cambiar su estado y apagarlo o encenderlo a demanda.

  Este servicio pretende demostrar con esta funcionalidad que es posible aplicar la solución a cualquier otro dispositivo dentro del ámbito de la IoT.

El lenguaje de programación seleccionado para el desarrollo de la solución es Java ya que cuenta con librerías y entornos de trabajo que facilitan mucho la implementación de los microservicios. Spring Boot y Spring Cloud son un ejemplo de estos frameworks que están principalmente orientados al desarrollo de servicios e integran de manera fácil dependencias como Netflix OSS que permiten aplicar patrones de comunicaciones para el descubrimiento de servicios o enrutamiento de solicitudes tal como se han visto en el punto anterior.

Utilizar estos entornos y librerías favorece que en la implementación de los microservicios se puedan reutilizar componentes ya desarrollados por terceros y que puedan integrarse con facilidad a la solución para poder reducir la complejidad y poder centrarse en el cumplimiento de los objetivos del proyecto.

4.4 Patrones arquitectónicos para la gestión de contenedores

4.4.1. Servicio por Contenedor

Uno de los puntos clave para desarrollar una solución utilizando este tipo de arquitecturas es la gran variedad de tecnologías que se pueden utilizar para desarrollar las funciones del sistema. Esto facilita poder utilizar diferentes alternativas ya que en ocasiones es más sencillo la implementación de algunas funciones en unos lenguajes de programación que en otros y, sobre todo, cuando se utilizan librerías especializadas que son externas y especificas al lenguaje.

La utilización de diferentes métodos de desarrollo implica también la necesidad de contar con diferentes entornos de ejecución y es en este aspecto donde los contenedores son una tecnología clave para esta arquitectura. Un contenedor permite la ejecución de diferentes aplicaciones desarrolladas en casi cualquier lenguaje de programación simplificando la gestión de dependencias y los entornos de ejecución necesarios.

El patrón Servicio por Contenedor define adoptar una arquitectura donde cada uno de los servicios se despliega en su propio contenedor y entorno de ejecución. Esto permite que los recursos gastados por cada uno de los servicios se puedan delimitar de forma personalizada y permite garantizar el máximo aislamiento posible en un entorno que se vuelve confiable.

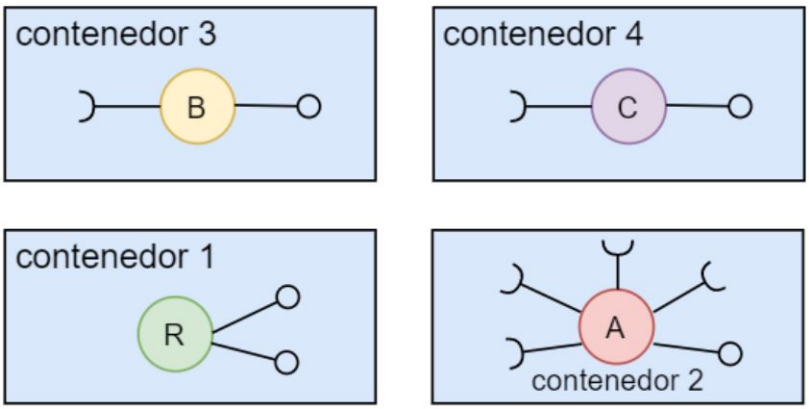


Figura 4.4: Modelo conceptual de servicio por contenedor

Algunas de las ventajas más destacadas de aplicar este patrón representado en la figura 4.4 son:

- La definición de un servicio en un contendor permite poder escalar hacia arriba y hacia debajo de forma sencilla pudiendo replicar una funcionalidad en concreto.
- Se trata de un entorno aislado donde solo se ejecuta un servicio en un entorno de ejecución personalizado con todas sus dependencias, por lo que no se ve afectado por resto de servicios y su comportamiento siempre es el mismo independientemente de las veces que se despliegue o retire.
- Utilizar encapsulación permite que los tiempos de puesta en marcha de un servicio sean muy rápidos lo que permite ajustar la elasticidad de forma casi inmediata antes un pico de carga puntual.

4.4.2. Patrón *sidecar*

La arquitectura establece una separación de las funciones de una aplicación en múltiples servicios y como se ha visto en el punto anterior, es posible aplicar patrones arquitectónicos donde se define que cada uno de estos servicios se ejecuten en un contenedor independiente.

Sin embargo, es habitual que varios servicios requieran funcionalidades de apoyo adicionales y comunes para realizar acciones como la configuración de red en el nodo donde se está ejecutando el contendor. Estas funcionalidades de apoyo no tiene sentido desarrollarlas dentro de la lógica principal del servicio ya que se estaría interfiriendo con su implementación funcional original y se repetiría una configuración adicional que es común en cada uno de los servicios que requieran soporte.

El patrón *sidecar* reafirma que esta no es la manera de proceder ya que por naturaleza estos servicios dentro de contenedores deben de únicamente preocuparse de proporcionar su funcionalidad y hacerlo de forma correcta sin interferencias. El patrón permite definir una separación donde el servicio principal se ubica en un contenedor como estaba previsto originalmente y la funcionalidad de apoyo en otro contenedor llamado *sidecar* dentro del mismo nodo.

Este nuevo contenedor permite ampliar las funcionalidades del servicio ejecutandose de manera paralela dentro del mismo nodo compartiendo una fuente de recursos común como el acceso a la persistencia de datos o a las interfaces de red sin afectar a la implementación original del servicio. Así mismo, es posible desplegar el *sidecar* de forma sencilla en cualquier otro nodo realizando cambios mínimos en su configuración.

Cuando se dispone de un nodo con capacidades limitadas y cuyos recursos no permiten ejecutar dos contenedores a la vez, una estrategia para la configuración de las interfaces de red puede ser la de lanzar en primer lugar un contenedor *sidecar* que se encargue de realizar esas operaciones y cuando estén listas, se retire para dejar paso al contenedor original que se desplegará con la configuración ya creada. En la figura 4.5 se puede ver un ejemplo de este tipo de estrategia.

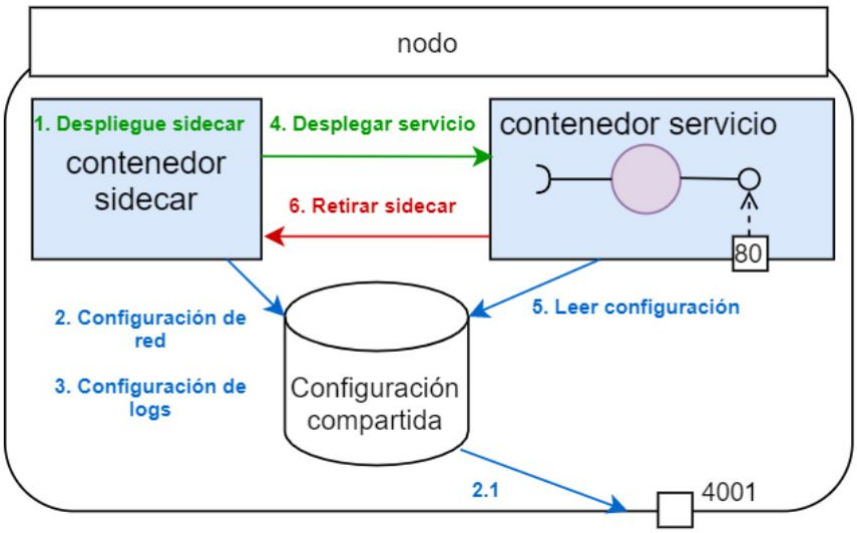


Figura 4.5: Modelo conceptual del patrón *sidecar*

4.5 Diseño del entorno de ejecución utilizando contenedores

El entorno escogido para la ejecución de los servicios son los contenedores que reducen la complejidad de gestionar las dependencias y librerías durante el despliegue de los microservicios. La tecnología seleccionada para el despliegue de estos contenedores es Docker al tratarse de la implementación de contenedores más extendida y soportada en el mercado y de ofrecer implementaciones propias que facilitan enormemente la creación de arquitecturas escalables y distribuidas.

La configuración de red que se utiliza para la comunicación entre los diferentes microservicios es transparente para los contendores ya que la única información que conocen es la interfaz de la propia red que tienen asociada. Para habilitar una comunicación con el resto es necesario publicar la disponibilidad de un puerto de forma explícita durante su creación o ejecución que permite relacionar este nuevo puerto virtual con uno existente en el nodo desde donde se ejecuta el contendor.

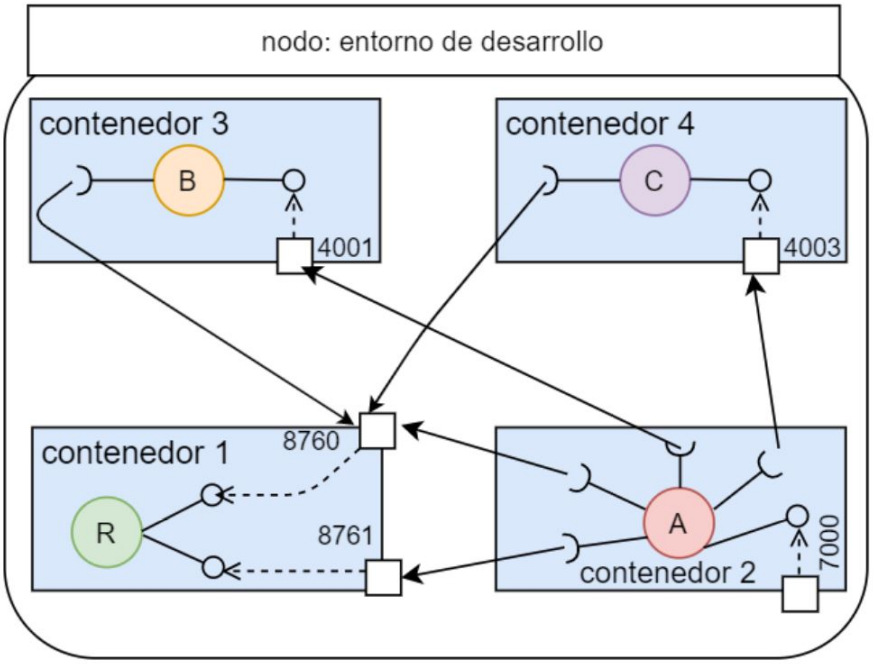


Figura 4.6: Diseño del entorno de ejecución con contenedores

En la figura 4.6 puede verse una representación de la solución de microservicios definida en este mismo capítulo ejecutándose en diferentes contenedores estructurados bajo un patrón de service per container. La funcionalidad se presenta de la siguiente forma:

- El 'contenedor 1' ejecuta el Registrador de servicios que proporciona su interfaz de registro bajo el puerto 8760 y su interfaz de consulta de localizaciones y recursos bajo el puerto 8761.
- El 'contenedor 2' ejecuta el API Gateway que proporciona la puerta de enlace a la solución bajo el puerto 7000.
- El 'contendor 3 y 4' presentan sus funcionalidades al resto de la solución bajo sus puertos 4001 y 4003 respectivamente.

4.6 Patrones arquitectónicos para el despliegue en infraestructura elástica en la nube

4.6.1. Balanceador de carga elástico

Como se ha definido durante el diseño de la solución de microservicios, el único punto de entrada para poder comunicarse con el sistema es a través del API Gateway, que es el encargado de gestionar las solicitudes redirigiéndolas a los servicios correspondientes o actuando como intermediario dentro de la propia red privada que utilizan los servicios.

La utilización de este patrón en exclusividad supone que el sistema dispone de un único punto de fallo (single point of failure) mediante el cual un fallo en su funcionamiento puede provocar que la solución de microservicios pueda quedar aislada e inoperativa de cara al usuario. Redundar y replicar este componente es fundamental para conformar un sistema de alta disponibilidad y garantizar que la solución siga funcionando a pesar de que uno de los componentes falle.

El patrón del balanceador de carga elástico (elastic load balancer) permite gestionar y distribuir de forma automática el acceso al sistema a través de una dirección de red pública que redirige las peticiones de los usuarios entre las múltiples instancias redundadas y disponibles del API Gateway aplicando una política de balanceado de carga.

Existen diferentes formas de que el balanceador de carga conozca la ubicación de las instancias redundas, ya bien sea delimitando que solo puedan desplegarse bajo direcciones que ya son conocidas de antemano por el balanceador o disponiendo de un registrador de servicios a quien consultar la ubicación. En la siguiente figura 4.7 se muestra una ejemplificación de este patrón.

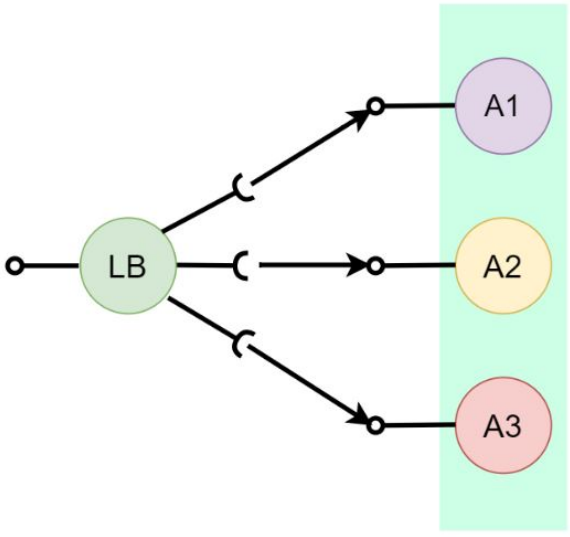


Figura 4.7: Modelo conceptual del balanceador de carga elástico

Un ejemplo es el de AWS Elastic Load Balancer (ELB) que permite equilibrar la carga de un trafico externo de Internet o de uno interno cuando se implementa en un VPC. Las instancias se agrupan mediante target groups que incluso pueden configurarse dentro de grupos de autoescalado.

4.6.2. Descubrimiento de servicios

Utilizar recursos de cómputo sin servidor libera al usuario de la configuración y gestión de la infraestructura para que pueda centrarse en otras cosas a cambio de perder la posibilidad de personalizar con detalle algunas características.

El proyecto plantea la necesidad de que los microservicios se ejecuten en diferentes localizaciones ya que pueden estar relacionados con dispositivos físicos y esto supone un problema a la hora del despliegue. Si se utilizan estos nodos no es posible realizar configuraciones especificas de red para permitir la comunicación con el resto de servicios a través de un espacio privado de comunicaciones.

La utilización de patrones de descubrimiento de servicios (*service discovery*) que se ha analizado anteriormente puede aportar también una solución a este problema. Algunos proveedores *cloud* proporcionan una implementación de este patrón proporcionando un servicio que permite descrubrir multiples recursos virtuales entre los que se incluyen microservicios, aunque para el diseño definitivo no es necesario utilizarlo ya que se gestiona a nivel de aplicación mediante un servidor de registros.

Lo importante es que también facilitan un descubrimiento a nivel de nodo de cómputo, proporcionando mecanismos para la creación de un espacio de red privado para que diferentes nodos que están distribuidos en diferentes zonas de disponibilidad sean alcanzables entre ellos. Esta opción sí que es interesante ya que resuelve de forma externa y eficaz la configuración que no era posible personalizar en las soluciones de computación *serverless*. En la figura 4.8 puede verse una representación del funcionamiento de este modelo aplicado a una solución de microservicios.

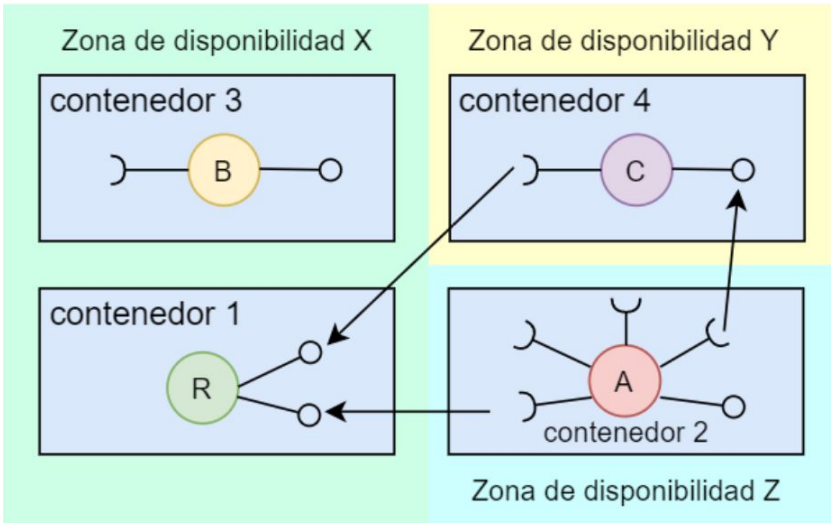


Figura 4.8: Modelo conceptual del descubrimiento de servicios

Un ejemplo de estos servicios de descubrimiento es el de AWS CloudMap que permite identificar con un nombre la localización de cualquier tipo de recurso virtual como bases de datos, aplicaciones *serverless*, instancias virtuales, colas de mensajerías o soluciones de microservicios.

4.7 Despliegue en infraestructura elástica en la nube

El despliegue de los contenedores en infraestructura elástica en al nube posibilita entre otras cosas utilizar las herramientas de orquestación que proporcionan los proveedores *cloud* y que facilitan la gestión y las comunicaciones dentro de un *cluster* de contenedores. Estas herramientas, además proporcionan los mecanismos necesarios para distribuir y replicar los contenedores entre diferentes zonas de disponibilidad consiguiendo que la solución disponga de una opción de ser tolerante a fallos.

La aproximación utilizada durante las primeras fases del diseño de un servicio por contenedor también se mantiene en este punto y se extiende de igual forma ubicando que cada contenedor se ejecute en un solo nodo de cómputo. La computación *serverless* facilita el aprovisionamiento de los recursos virtuales que son necesarios de forma automática y económica para estos contenedores, por lo que se aprovecha la máxima capacidad de cómputo sin llegar a caer en un sobredimensionamiento difícil de gestionar con instancias de máquinas virtuales.

No obstante, esta estructuración de la solución implica la utilización de patrones *sidecar* para evitar implicar a los servicios sobre la configuración de la infraestructura y a la vez configurar los aspectos de red que son necesarios para la comunicación de los microservicios.

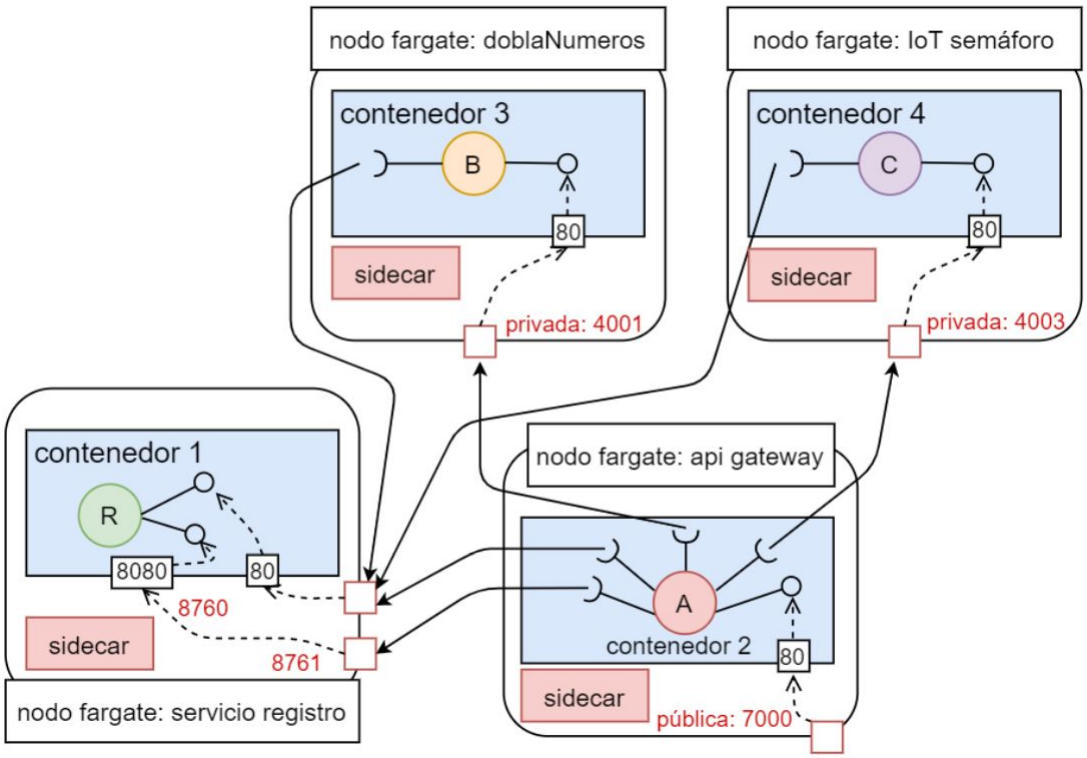


Figura 4.9: Configuración de red antes del despliegue en la nube

En la figura 4.9 puede verse una representación de la solución de microservicios definida y que ha sido desplegada para ejecutarse en contenedores dentro de una infraestructura de nodos que utilizan tecnología computación *serverless*.

Estructurar el sistema para que cada contenedor disponga de su propio nodo de computo implica la necesidad de imponer un orden en el despliegue de los contenedores, ya que es necesario primero levantar el *sidecar* para que realice la configuración de red en las interfaces del nodo para luego retirarlo y desplegar el contenedor final de servicio.

Así mismo, también se toma la decisión de modificar la implementación de los microservicios y contenedores para delegar al nodo el requerimiento de presentar las funcionalidades en puertos aleatorio con motivo de conseguir escalabilidad en el sistema. Esta decisión en el diseño simplifica aún más el desarrollo de los microservicios y contenedores evitando posibles errores de implementación.

El proveedor *cloud* elegido para el despliegue de la solución es Amazon Web Services (AWS) ya que ofrece el servicio ECS con una herramienta de orquestación de contenedores más sencilla de configurar y poner en marcha que la alternativa en Kubernetes y que además está perfectamente integrada con el resto de los servicios de su plataforma y presentando una tarificación de utilización más económica.

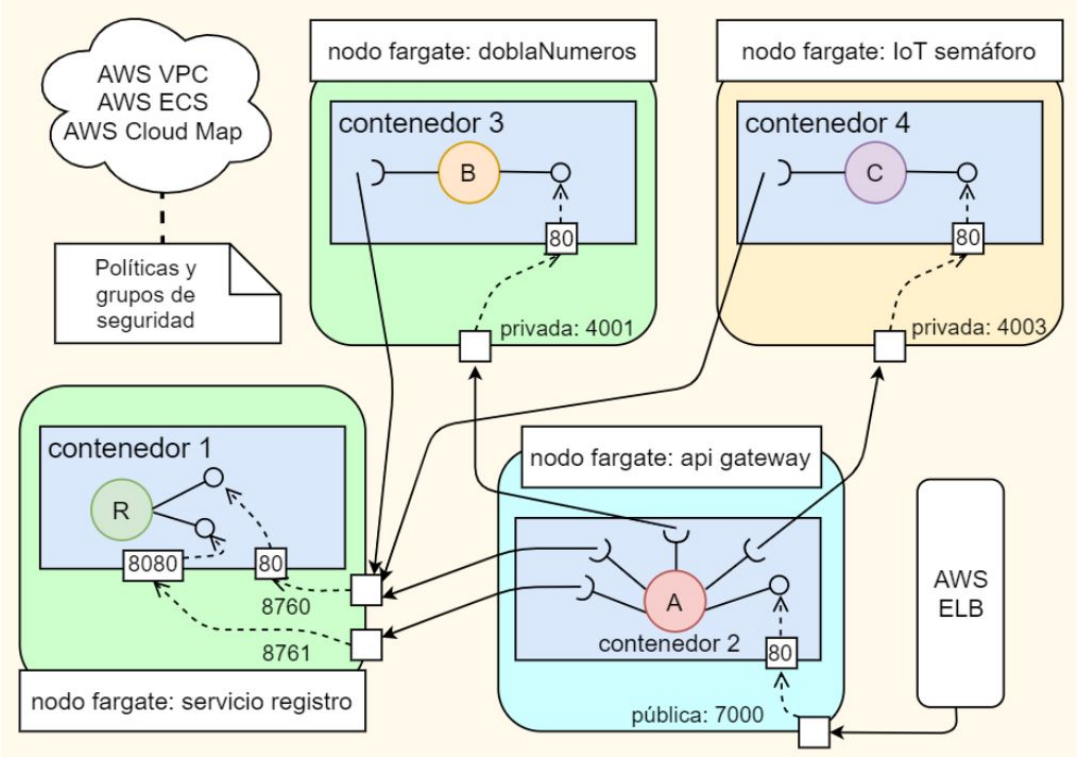


Figura 4.10: Despliegue de la solución en infraestructura elástica en la nube de AWS

En la figura 4.10 puede verse una representación de la solución ejecutándose en el *cluster* de contenedores de ECS utilizando los nodos de cómputo AWS Fargate. Algunos de los detalles definidos durante el diseño del despliegue son:

- Se utiliza el patrón de descubrimiento para hacer posible que sean alcanzables algunos de los nodos de cómputo que están distribuidos en zonas de disponibilidad distintas y representadas por los colores verde, amarillo y azul.
- Se posiciona un balanceador de carga elástico en la solución para distribuir de forma automática el acceso al sistema a través de una dirección de red pública que redirige las peticiones al nodo que contiene el servicio del API Gateway.

4.8 Diseño de la solución *adaptive-ready*

Una solución *adaptive-ready* es aquella solución sobre la que se ha desarrollado una capa de adaptación compuesta por sensores y efectores con el objetivo de conseguir que un sistema administrado esté preparado para poder autogestionarse de forma autónoma. Estos sensores y efectores recopilan información y ejecutan operaciones sobre los servicios de un sistema administrado realizando acciones que implican desplegar la solución desde cero o reconfigurar su arquitectura lógica.

La nueva capa de adaptación *touchpoint* 'punto de contacto' sigue el modelo definido en el artículo [26] que define la arquitectura de referencia para el desarrollo de soluciones auto-adapatativos.

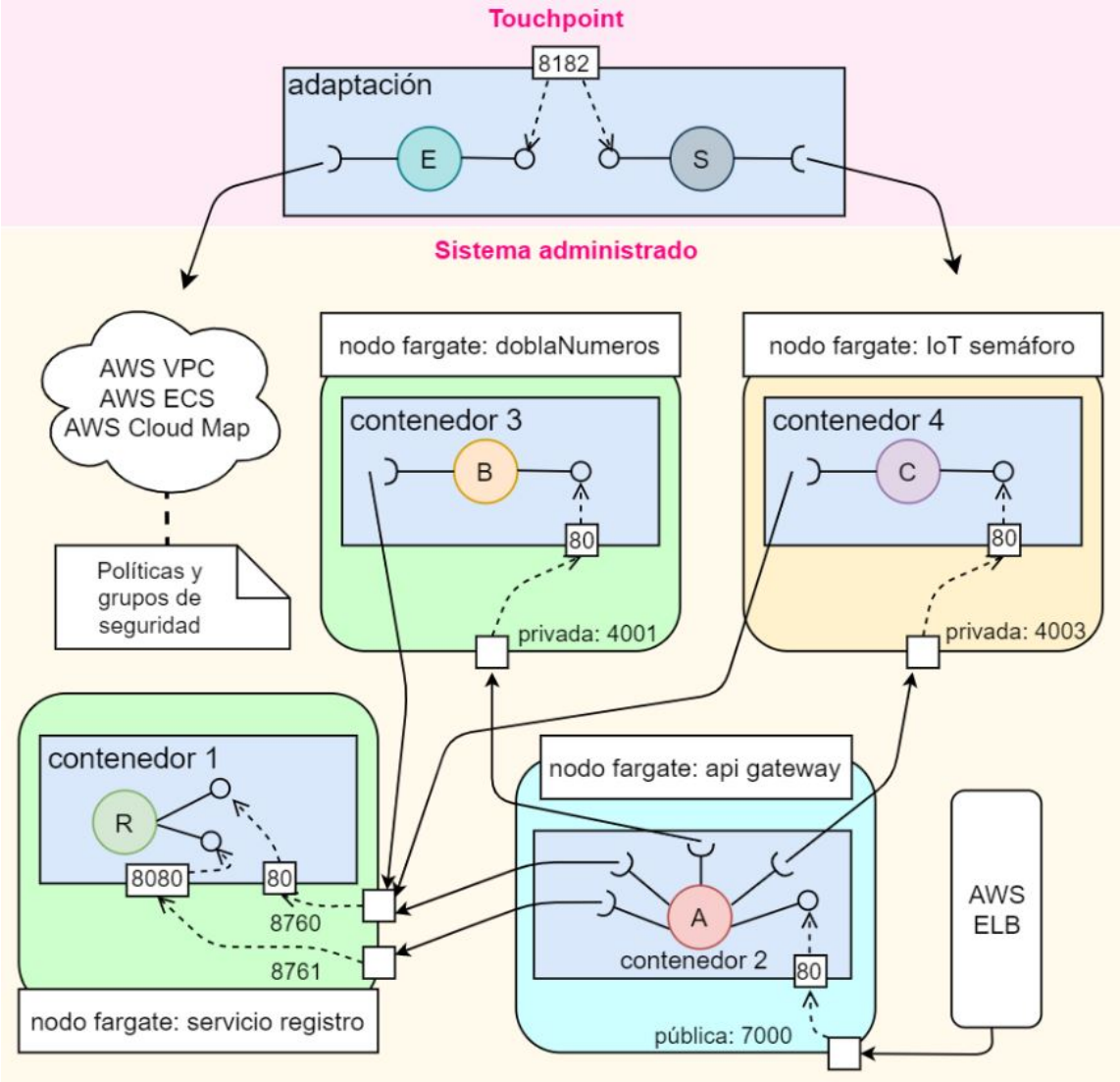


Figura 4.11: Diseño de los componentes de adaptación

En la figura 4.11 se puede ver una representación de la solución de microservicios desplegada sobre AWS y el nuevo componente de adaptación situado encima del sistema. La funcionalidad de este nuevo componente esta expuesta a través de una interfaz API REST accesible desde el puerto 8182 sobre la que el administrador de autonomía puede realizar peticiones.

A continuación, se listan la acciones que el componente de adaptación debe de ser capaz de ejecutar a través de su efector y exponer a través de la interfaz API REST:

- Crear un *cluster* de contenedores.
- Crear infraestructura y recursos de red asociados al *cluster*.
- Crear políticas y grupos de seguridad para la definición de redes de contenedores privadas.
- Desplegar balanceadores de carga elásticos y servicios de descubrimiento de nodos entre zonas de disponibilidad.
- Consultar el estado de cada uno de los servicios y su configuración de despliegue.
- Desplegar un nuevo servicio en el *cluster* de contenedores a partir de una imagen Docker.
- Retirar un servicio del *cluster* eliminando y cancelando su registro en el servicio registrador de servicios.
- Reconfigurar la configuración de un servicio desplegado.

Dada la complejidad que está tomando el proyecto como puede verse a lo largo del diseño de la solución y la cantidad de componentes que se tienen que desarrollar e integrar tanto en el sistema administrado como en la adaptación, es necesario volver a plantear el alcance del proyecto y acotar la actuación sobre la solución.

El nuevo alcance plantea el diseño realizado hasta ahora, donde la solución que se ejecuta sobre infraestructura en la nube ya es ***adaptive-ready*** y está preparada para que sea manejada por un tercer componente (que no se implementará) con el rol de administrador de autonomía y que permite convertir la solución en auto adaptativa.

4.9 Despliegue automatizado con herramientas infraestructura como código

Para convertir la solución en *adaptive-ready* es necesario poder gestionar la infraestructura en la nube desde el nuevo componente ya que requiere poder ejecutar acciones para desplegar y configurar nuevos recursos virtuales, retirar los ya existentes o monitorizar su actividad y estado. Todas estas operaciones deben de ejecutarse de forma programática cuando reciba una solicitud a través de su API REST en el puerto 8182.

Acciones	Recursos AWS	Parámetros de entrada	Parámetros de salida
Configuración de red	Amazon VPC VPC subnets con diferentes zonas de disponibilidad Espacio de nombres privado en AWS CloudMap Grupo de Seguridad por defecto (SG)		Id. VPC Id. Subnets Id. CloudMap
Despliegue de un *cluster* de contenedores	*Cluster* de ECS Grupo de Seguridad asociado al *cluster* (SGC) Balanceador de carga elástico (ELB)	Id. VPC Id. SG	Id. *Cluster* Id. SG *cluster* Id. ELB IP pública ELB
Despliegue de un servicio	Definición de tarea (nodo) en AWS Fargate Target group asociado al ELB Listener asociado al ELB Configuración de red en el SGC Definición del *sidecar* en ECS Definición del servicio en ECS con imagen docker	Imagen *sidecar* Imagen contenedor Id. *Cluster* Id. SG *cluster* Id. subnets Id. ELB Id. CloudMap	Id. Tarea Fargate Id. Servicio

Tabla 4.4: Recursos necesarios para automatizar el despliegue en AWS

Las herramientas de infraestructura como código (IaC) son indispensables para poder abordar este requerimiento ya que posibilitan administrar, monitorizar y aprovisionar recursos de forma automática mediante la definición de recetas. Cloudformation es la implementación de una de estas herramientas que ha sido especialmente desarrollada por Amazon para la definición y gestión de sus recursos virtuales en AWS.

En la tabla 4.4 se han identificado los recursos imprescindibles para poner en marcha la solución y cuáles son los parámetros de entrada y salida que necesitan para transformar el despliegue que hasta ahora venía siendo manual en uno automatizado. Además, se estructura las operaciones de aprovisionamiento en tres tipos de recetas distintas que formarán las pilas de recursos virtuales durante el despliegue para controlar las dependencias entre ellos.

El orden de ejecución para cada una de estas acciones de aprovisionamiento se muestra a continuación.

Configuración de red

1. Creación del Amazon VPC.
2. Creación de las subnets dentro del VPC con diferentes zonas de disponibilidad.
3. Crear un espacio de nombre privado en AWS CloudMap.
4. Crear un grupo de seguridad (SG) general.

Despliegue de un *cluster* de contenedores

1. Crear un *cluster* de contenedores en ECS.
2. Crear un grupo de seguridad (SG) para la comunicación dentro del *cluster*.
3. Asociar el grupo de seguridad del *cluster* al general.
4. Crear un balanceador de carga elástica (ELB) con una dirección IP pública.

Despliegue de un servicio

1. Crear una definición de tarea (nodo) en AWS Fargate.
2. Desplegar un contenedor *sidecar* para configurar las interfaces de red.
3. Retirar el contenedor *sidecar*.
4. Desplegar un contenedor con la imagen Docker indicada como entrada.
5. (opt.) Crear un Target Group en el ELB si la interfaz de red es pública.
6. (opt.) Asociar el nodo Fargate a un Listener dentro del target Group del ELB.

La ejecución de estas recetas y la monitorización de los recursos ya desplegados en la plataforma requiere utilizar la API de Java para AWS, AWS Java Software Development Kit (SDK) v2, e integrarla dentro de la implementación de los sensores y efectores.

4.10 Tecnologías utilizadas

Solución	Tecnología	Descripción
Spring Boot	Desarrollo de microservicios	Es un framework open source basado en el lenguaje de programación Java que permite implementar microservicios con una comunicación distribuida y desacoplada.
Netflix OSS	Arquitecturas y patrones de aplicaciones distribuidas	Es un conjunto de librerías y soluciones desarrolladas por Netflix que permite resolver problemas de escalación de sistemas distribuidos utilizando patrones arquitectónicos.
Docker	Contenedores	Es el sistema de contenedores más utilizado y extendido que facilita el desarrollo y despliegue de aplicaciones. Soportado por las principales herramientas de orquestación y soluciones *cloud*.
Amazon Web Services (AWS)	Servicios en la nube	Uno de los principales proveedores de servicios en la nube del mercado que ofrece soluciones de infraestructura elástica de computo, networking, almacenamiento y seguridad a través de un modelo pago por uso.
Elastic Container Service (ECS)	Orquestación de contenedores	Una de las principales herramientas de orquestación de contenedores en AWS que facilita la ejecución de soluciones en un *cluster* de nodos distribuidos entre varias zonas de disponibilidad.
AWS Fargate	Computación *Serverless*	Solución de computación sin servidor orientada a la ejecución de contenedores ofrecida por AWS y que simplifica el aprovisionamiento de recursos virtuales de cómputo de forma automática sin tener que especificar sus características.
Restlet	API REST	Una de las principales herramientas de desarrollo para la implementación de interfaces API REST a través de un framework Java.
Cloudformation	Infraestructura como código	Herramienta de despliegue de infraestructura como código desarrollada para funcionar sobre AWS y que aprovisiona recursos virtuales a través de recetas escritas en JSON o YAML.
AWS SDK for Java 2.X	Cloud SDK	Herramientas SDK para el desarrollo de software que proporciona una API para el lenguaje de programación Java que permite interactuar con los recursos desplegados en la nube de AWS.

Tabla 4.5: Tecnologías definidas para la implementación durante el diseño de la solución

CAPÍTULO 5
Implementación y despliegue

5.1 Implementación de los microservicios

La implementación de la solución utiliza los frameworks Spring Boot y Spring Cloud para Java ya que facilitan el uso de herramientas que son necesarias para abordar el desarrollo de arquitecturas basadas en microservicios y afrontar los típicos problemas de los entornos distribuidos.

Spring Initializr es una de estas herramientas web que proporciona una interfaz para la configuración inicial del proyecto que simplifica enormemente el trabajo previo sobre la gestión de las dependencias requeridas.

Su función es la verificación automática de la compatibilidad entre todas las dependencias seleccionadas, así como sus diferentes versiones, tipo de entorno de ejecución escogido y las versiones disponibles de Java soportadas. El resultado es un proyecto disponible para descargar con toda la estructura de carpetas ya creada y el archivo de compilación con las dependencias incluidas.

5.1.1. Servidor de registro de servicios

Eureka es una herramienta que proporciona la funcionalidad de un servidor de registros a un servicio (Eureka Server) a través de una interfaz API REST. El resto de los servicios actúan como clientes (Eureka Clients) comunicándose a través de esta interfaz con el servidor para comunicar las actualizaciones relacionadas con su estado.

La integración de la funcionalidad de servidor es sencilla y consiste en añadir una notación (@EnableEurekaServer) al servicio implementado con Java y que va a adoptar ese rol.

```
1 @EnableEurekaServer
2 @SpringBootApplication
3 public class Microservicio {
4
5  public static void main(String[] args) throws Exception {
6     SpringApplication.run(Microservicio.class, args);
7  }
8
9 }
```

La configuración de este servidor implementado con la dependencia "Eurkea Server", se realiza mediante el archivo de propiedades de la aplicación y mantiene la siguiente estructura:

```
# Nombre del servicio
spring:
  application:
    name: servidor-eureka

# API Rest Server
server:
 port: 8761

# Configuraci n de Eureka
eureka:
 client:
  #Se deshabilita su propio registro como servicio
  registerWithEureka: false
```

Los clientes que utilicen Java como *runtime environment* pueden integrar de forma fácil una instancia de Eureka Client a través de la dependencia "Eureka Discovery Client". Es suficiente con añadir una anotación (@EnableDiscoveryClient) al servicio y realizar una configuración en el archivo de propiedades de la siguiente forma.

```
# Nombre del servicio
spring:
  application:
    name: cliente-eureka

# Configuraci n de Eureka
eureka:
  client:
    serviceUrl:
      defaultZone: #direcci n del eurkea server
```

Un procedimiento alternativo que permita a los clientes que no utilicen Java como lenguaje de implementación es el de realizar una petición de registro a través de un método POST a la interfaz API REST del Eureka Server. Un ejemplo de mensaje de registro tipo en formato JSON se muestra a continuación.

```
{
    "instance": {
        "hostName": "localhost",
        "app": "servicio-ejemplo",
        "vipAddress": "servicio-ejemplo",
        "ipAddr": "10.9.8.7",
        "status": "UP",
        "port": {
            "$": 4001,
            "@enabled": true
        },
        "dataCenterInfo": {
            "@class": "",
            "name": ""
        }
    }
}
```

5.1.2. Servicio API Gateway

Zuul es una herramienta que proporciona una implementación para integrar un servidor API Gateway en un microservicio. Permite proporcionar una puerta de enlace donde se gestionan todas las solicitudes entrantes del sistema y realizar un enrutamiento dinámico hacia los microservicios destino actuando como intermediario para resolver las peticiones.

Con un funcionamiento similar al de los Edge Servers puede configurarse para dar soporte de autenticación y seguridad, gestión de un balanceao de carga entre microservicios o generar respuestas estáticas.

La integración de la funcionalidad de servidor consiste en añadir una notación (@EnableZuulProxy) al servicio implementado con Java y que va a adoptar el rol.

```
@EnableZuulProxy
@SpringBootApplication
public class Microservicio {

 public static void main(String[] args) throws Exception {
    SpringApplication.run(Microservicio.class, args);
 }

}
```

El servidor Zuul se apoya en el registro de servicios implementado por Eureka Server para localizar los recursos disponibles y proporcionar esa funcionalidad de enrutamiento dinámico. A continuación, se muestra la estructura tipo del archivo de propiedades de la aplicación que contiene la configuración de Zuul.

```
# Nombre del servicio
spring:
  application:
    name: servidor-zuul

# API Rest Server
server:
 port: 7000

# Configuraci n de Eureka
eureka:
  client:
    serviceUrl:
      defaultZone: #direcci n del eurkea server

# Configuraci n de Zuul
zuul:
  routes:
      serv-greet:
        path: /nombre-servicio/**
        serviceId: #id del servicio en Eureka
```

5.1.3. Servicio controlador de semáforo

La desarrollo de este servicio tiene el objetivo de demostrar que la solución de microservicios es lo suficientemente genérica para que pueda ser aplicable a cualquier ámbito de la Internet de las Cosas y de la Industria 4.0 y por este motivo su implementación puede tener una aplicación práctica y directamente relacionada con un dispositivo físico.

El dispositivo físico consiste en un semáforo con sus tres luces (roja, ámbar y verde) sobre el que ya hay desarrollado una solución IoT con una Raspberry Pi que expone de forma nativa una serie de operaciones para interactuar con él a través de una API REST o una cola de mensajes MQTT.

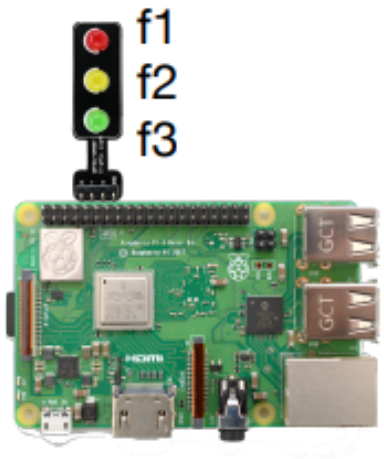


Figura 5.1: Representación del dispositivo físico

La implementación de este microservicio extiende su funcionalidad integrándolo dentro de la solución *adaptive-ready* desplegada en infraestructura en la nube. Algunas de estas nuevas funciones disponibles vienen dadas por el resto de los componentes que ya han sido definidos como por ejemplo una monitorización de su disponibilidad por parte del registrador de servicios o la publicación de su interfaz API nativa a través del API Gateway que le proporciona de extra adicional de seguridad o funcionalidades de balanceo de carga.

Otra opción interesante que no se va a implementar pero que es posible alcanzar integrando el dispositivo en la plataforma es la de desarrollar su digital twin, para mantener una copia virtual de su estado con la que poder comunicarse en el caso de que el dispositivo físico no esté disponible y se pueda seguir prestando servicio.

Las operaciones que se exportan y se publican desde la interfaz API REST del API Gateway son las siguientes.

- **GET. Consultar el estado del semáforo**

  Para consultar el estado actual del semáforo es necesario realizar una petición al siguiente recurso.

```
api-gateway:7000/traffic-light
```

  La respuesta que debería de obtenerse es la siguiente.

```
{
    "id":"f1",
    "estado":"Encendido"
},
{
    "id":"f2",
    "estado":"Apagado"
},
{
    "id":"f3",
    "estado":"Parpadeando"
}
```

- **GET. Obtener el puerto del servicio del semáforo**

  Para consultar el puerto en el que se ejecuta el servicio es necesario realizar una petición al siguiente recurso.

```
api-gateway:7000/traffic-light/port
```

  La respuesta que debería de obtenerse es la siguiente.

```
{
    "port": 1234
}
```

- **PUT. Cambiar el estado del semáforo**

  Para encender o apagar el semáforo es necesario realizar una petición al siguiente recurso.

```
api-gateway:7000/traffic-light/port
```

  Si se quiere encenderlo, se acompaña del siguiente cuerpo de mensaje.

```
{
    "accion":"activar"
}
```

  Si se quiere apagarlo, se acompaña del siguiente cuerpo de mensaje.

```
{
    "accion":"desactivar"
}
```

La respuesta que debería de obtenerse si la operación se ha ejecutado correctamente es la siguiente, acompañado de un 200 OK.

```
{
}
```

- **GET. Consultar el estado de una función del semáforo**

  Para consultar el estado actual del semáforo es necesario realizar una petición al siguiente recurso.

```
api-gateway:7000/traffic-light
/{nombre de la funcion}
```

  Teniendo en cuenta que el semáforo sigue la siguiente relación de funciones:

  - F1 = Luz Roja
  - F2 = Luz Ámbar
  - F3 = Luz Verde

  Para consultar el estado actual de la luz roja, el recurso al que realizar la petición sería.

```
api-gateway:7000/traffic-light/f1
```

  La respuesta que debería de obtenerse es la siguiente.

```
{
    "id":"f1",
    "estado":"Encendido"
}
```

- **PUT. Cambiar el estado de una función del semáforo**

  Para encender o apagar una del semáforo es necesario realizar una petición al siguiente recurso que como ejemplo se escoge la luz roja.

```
api-gateway:7000/traffic-light/f1
```

  Si se requiere encenderla, se acompaña del siguiente cuerpo de mensaje.

```
{
    "accion":"encender"
}
```

  Si se requiere apagarla, se acompaña del siguiente cuerpo de mensaje.

```
{
    "accion":"apagar"
}
```

Si se requiere que parpadeé, se acompaña del siguiente cuerpo de mensaje.

```
{
    "accion":"parpadear"
}
```

5.1.4. Servicio multiplicador de números

El desarrollo de este servicio implementa funcionalidades arquetípicas que tienen como objetivo incorporar un microservicio más dentro de la solución para demostrar que es posible desplegarse y retirar tantos como se requiera sin afectar al funcionamiento del sistema.

Por este motivo y para no añadir más complejidad a la implementación, la funcionalidad proporcionada a través de una interfaz API REST consiste en un multiplicador de números. El servicio recibe como entrada un número que devuelve multiplicado por dos y además añade en la respuesta un contador con el número de peticiones que ha resuelto ya al usuario.

Las operaciones accesibles desde el API Gateway se muestran a continuación.

- **GET. Multiplicar el numero pasado como parámetro por dos**

  Para consultar el resultado de multiplicar un número por dos es necesario realizar una petición al siguiente recurso.

```
    api-gateway:7000/multiplier/double?number={introducir numero}
    "api-gateway:7000/multiplier/double?number=3"
```

  La respuesta que debería de obtenerse es la siguiente.

```
{
    "id": 1,
    "result": "El resultado es 6.
}
```

- **GET. Obtener el puerto del servicio multiplicador**

  Para consultar el puerto en el que se ejecuta el servicio es necesario realizar una petición al siguiente recurso.

```
    api-gateway:7000/multiplier/port
```

  La respuesta que debería de obtenerse es la siguiente.

```
{
    "port": 1234
}
```

5.2 Creación de las imágenes de contenedor

Una vez finalizada la implementación de los microservicios es cuando empieza el proceso de preparación para que se ejecuten en contenedores. El sistema de contenedores seleccionado para realizar este despliegue es Docker al ser la solución más extendida y soportada por las principales herramientas de orquestación y soluciones *cloud* que se utilizarán más adelante.

Antes de continuar es necesario aclarar los siguientes conceptos con respecto a Docker.

- **Imagen Docker**

  Una imagen Docker puede interpretarse como una plantilla para la creación de contenedores. Se sitúan como la imagen base a partir de la cual se ejecuta un contenedor que contiene el conjunto de instrucciones, entornos de ejecución, librerías, herramientas y otras dependencias necesarias para la ejecución de una aplicación.

- **Dockerfile**

  Un archivo Dockerfile puede interpretarse como un fichero de configuración para la creación de imágenes Docker. Habitualmente se utilizan también para ampliar la disposición y composición inicial de una imagen preexistente, sobre la que se añade una nueva serie de instrucciones, entornos de ejecución y librerías para la construcción de una nueva imagen personalizada.

- **Docker-compose**

  Un archivo Docker-compose puede interpretarse como una plantilla que define la creación y ejecución de una serie de contenedores al mismo tiempo y agrupados bajo una única solución que comparte una configuración en común.

  Es una herramienta utilizada habitualmente con arquitecturas de microservicios donde cada uno de los servicios está en un contenedor independiente y es necesario que se ejecuten como una única solución compartiendo una configuración de red común.

La preparación comienza con la creación de estos archivos Dockerfile que permitirán construir una imagen por cada microservicio. Un ejemplo de la implementación de un dockerfile puede verse a continuación.

```
1 FROM openjdk:8-jdk-alpine
2 ARG JAR_FILE=target/*.jar
3 COPY ${JAR_FILE} app.jar
4 ENTRYPOINT ["java","-jar","/app.jar"]
```

Luego es necesario proporcionarle al Dockerfile la ruta donde se encuentra el .jar de los servicios que se han desarrollado y ejecutar el siguiente comando para construir la imagen.

```
docker build \
--build-arg JAR_FILE=microservicio.jar \
-t carizmar/microservicio .
```

Todos los servicios se han implementado utilizando el lenguaje de programación Java y por este motivo se ha seleccionado como imagen base del contenedor una implementación open source de la plataforma, OpenJDK. La variante alpine proporciona un tamaño de imagen mucho más pequeño de lo normal por lo que facilita que la solución sea más flexible y el despliegue más rápido.

La siguiente parte consiste en la implementación de un Docker-compose para iniciar los microservicios con el objetivo de depurar la solución y comprobar que no existen errores durante la ejecución. Este desarrollo luego no será necesario ya que para el despliegue en la nube solo es necesario disponer de las imágenes de los microservicios.

```
version: '3'

services:

#Definici n del servidor de registro
registry-service:
    image: carizmar/service-registry
    container_name: service-registry
    #No es necesario exponer el puerto de eureka
    #pero se expone para verificar su funcionamiento en el desarrollo
    ports:
    - '8760:8760'
    networks:
    - 'internal-network'

#Definicion del api-gateway
apigateway-service:
    image: carizmar/api-gateway
    container_name: api-gateway
    ports:
    - '7000:7000'
    networks:
    - 'internal-network'

#Definicion del sem foro
trafficlight-microservice:
    image: carizmar/trafficlight-microservice
    container_name: trafficlight-microservice
    networks:
    - 'internal-network'

#Definicion del multiplicador
multiplier-microservice:
    image: carizmar/multiplier-microservice
    container_name: multiplier-microservice
    networks:
    - 'internal-network'

networks:
internal-network:
```

5.3 Implementación del componente adaptativo

El componente adaptativo plantea una estructura y arquitectura similar a la utilizada en el resto del proyecto utilizando una solución de microservicios compuesta por diferentes sensores y efectores que implementan esa capa *touchpoint* de 'punto de contacto' sobre el sistema administrado.

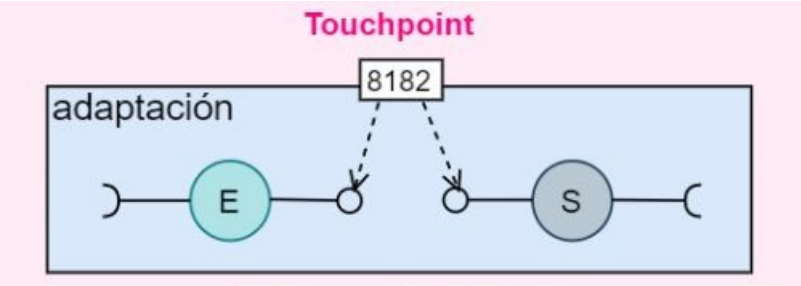


Figura 5.2: Implementación de sondas y efectores

Este nuevo componente también utiliza el lenguaje de programación Java para su implementación junto con una serie de librerías para a apoyar su funcionalidad. Las librerías más relevantes son:

- **AWS SDK for Java 2.X**

  Se trata del SDK que proporciona las herramientas para poder lanzar los *templates* de Cloudformation y monitorizar los recursos virtuales que ya están desplegados.

- **Restlet**

  Se trata de una herramienta que permite construir interfaces API REST más complejas y robustas que las que se estaban utilizando hasta ahora. Permite que el componente exponga de manera global las funcionalidades proporcionadas por los diferentes sensores y efectores.

El componente adaptativo implementa y estructura un espacio definido para el sistema administrado sobre el que controla el despliegue y el estado de los diferentes componentes que componen la solución. Los componentes del sistema pueden ser de dos tipos distintos:

- **Contenedores de microservicios**

  Estos componentes son los contenedores Docker que contienen las librerías y aplicaciones necesarias para poner en marcha un microservicio.

- **Recursos virtuales**

  Son los componentes que están relacionados con un servicio o recurso virtual de un proveedor *cloud* y son necesarios para poder iniciar los contenedores de microservicios.

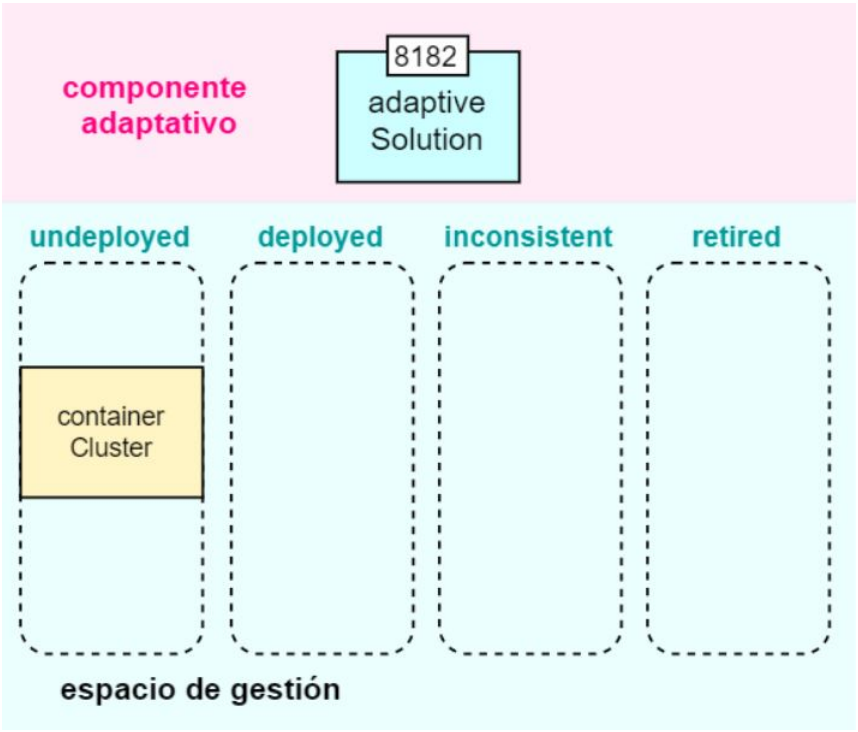


Figura 5.3: Estado inicial del espacio administrado de adaptiveSolution

En la figura 5.3 se puede ver una representación de la implementación del componente adaptativo denominado *'adaptive solution'* y su espacio administrado que ha definido para categorizar a los componentes de la solución dependiendo de su estado. Este modelo conceptual representa el estado inicial del sistema, donde un componente puede pasar por los siguientes estados.

- **Sin desplegar (Preparado)**

  El componente no está en ejecución, pero está dentro del espacio administrado por el controlador y preparado para ser desplegado en cualquier momento.

- **Desplegado**

  El componente esta desplegado y ejecutándose en la infraestructura del proveedor *cloud*. Durante este estado los componentes son monitorizados por los sensores de adaptación.

- **Inconsistente**

  El componente ha perdido la comunicación con los sensores de adaptación y no puede determinar cuál es su estado actual.

- **Retirado**

  Las sondas han detectado un comportamiento errático en el funcionamiento del componente por lo que se anula el despliegue y se retira para no volver a ejecutarlo.

Durante la puesta en marcha de la adaptación se ha implementado que ya exista un componente en el espacio administrado, aunque no esté desplegado. Este componente es el *cluster* de contenedores de AWS que lleva asociada una receta en Cloudformation para realizar la configuración de red previa y el aprovisionamiento de los recursos en ECS.

Alguna de las operaciones más relevantes que se pueden solicitar en la interfaz API REST se describen a continuación.

- **GET (Sensor). Consultar el estado del sistema**

  Para consultar el estado actual del sistema administrado es necesario realizar una petición al sensor del recurso adaptiveSolution.

```
sensor.localhost:8182/adaptiveSolution
```

  La figurar 5.3 representa el estado inicial del sistema donde aún no se han dado de alta los servicios ni desplegado el *cluster* de contenedores por lo que la respuesta que debería de obtenerse es la siguiente.

```
{
    "undeployed": [
        {
            "containerCluster": 0
        }
    ],
    "deployed": [],
    "id": "touchpoint",
    "inconsistent": [],
    "retired": []
}
```

  Cuando la solución ya tiene varios servicios desplegados la respuesta debería de ser similar a la siguiente:

```
{
    "undeployed": [
        {
            "apiGateway": 0
        }
    ],
    "deployed": [
        {
            "serverRegistry": 1
        },
        {
            "containerCluster": 1
        }
    ],
    "id": "touchpoint",
    "inconsistent": [
        {
            "multiplierService": 0
        }
    ],
    "retired": []
}
```

Una modelo conceptual de la respuesta obtenida se puede ver represado en la figura 5.4.

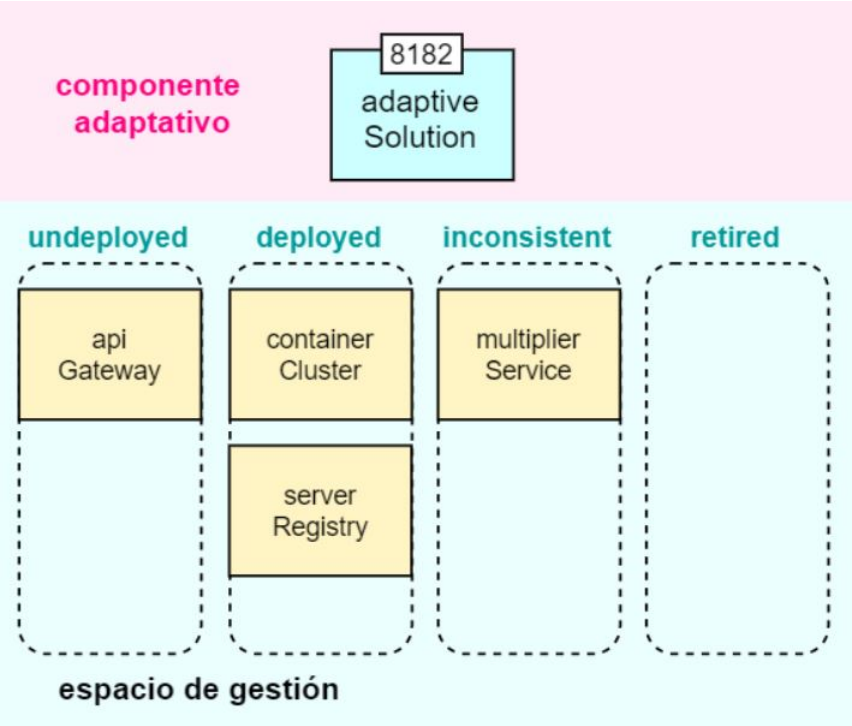


Figura 5.4: Servicios desplegados dentro del espacio administrado de adaptiveSolution

- **GET (Sensor). Consultar el estado de un componente del sistema administrado**

  Para consultar el estado actual de un componente es necesario realizar una petición al sensor del componente.

```
sensor.localhost:8182/adaptiveSolution/service
/{nombre del componente}
```

Siguiendo la situación representada en la figura 5.4, la ruta al recurso para hacer una consulta al componente *cluster* de contenedores sería:

```
sensor.localhost:8182/adaptiveSolution/service/containerCluster
```

El resultado que devolvería la consulta sería parecida a la siguiente:

```
{
    "template": "https://carizmar-tfm.s3.amazonaws.com/containerCluster.
        yml",
    "port": 0,
    "ip": "conta-LoadB-148J8FNZ5FF4N-29a742f11c5b399c.elb.us-east-1.
        amazonaws.com",
    "id": "containerCluster",
    "status": "DEPLOYED"
}
```

La respuesta a esta petición siempre está estructurada en los siguiente atributos:

- ***Template***
  Devuelve la plantilla Cloudformation que ha construido la adaptación y que se utiliza para el despliegue del componente. Es posible consultar la receta ya que almacena una copia en un recurso de Amazon Simple Storage Service (S3).

- **IP**
  Indica la dirección pública del Elastic Load Balancer desde donde es accesible el *cluster* que ejecuta la solución de microservicios.

- **Puerto**
  Indica el puerto asociado al ELB desde donde es accesible el recurso. El componente no es accesible si el valor obtenido es 0 como en el ejemplo.

- **Id**
  Nombre del componente dentro del sistema administrado.

- **Status**
  Indica el estado en el que se encuentra el componente.

- **POST (Efector). Añadir un nuevo componente al sistema administrado**

  Para añadir un nuevo componente al sistema es necesario realizar una petición al efector del recurso adaptiveSolution.

```
efector.localhost:8182/adaptiveSolution
```

Con el siguiente cuerpo acompañando la petición:

```
{
  "id": "trafficLight",
  "status": "undeploy",
  "port": 8761,
  "ip": "public",
  "image": "carizmar/service-registry"
}
```

Una petición para añadir un nuevo componente siempre estará estructurada por los siguiente atributos:

- **Id**
  Nombre del nuevo componente dentro del sistema administrado.

- **Status**
  Todos los componentes se añaden al sistema sin desplegar ya es necesario realizar una configuración previa. Una vez se haya realizado esta configuración y se haya integrado en la solución, se procederá a desplegarse si se indica el valor “deploy”.

- **IP**
  Si se le pasa el valor públic significa que el componente será accesible desde el ELB y durante la construcción de la receta de Cloudformation se configurarán los recursos de red necesarios como los Target Group y los Listeners.

  Si se le pasa el valor private significa que el componente no será accesible desde el ELB, pero sí desde el API Gateway si así se ha configurado en la implementación del microservicio.

- **Puerto**
  Indica el puerto asociado al ELB desde donde es accesible el recurso.

- **Image**
  El valor que se le pasa a este atributo es la localización de una imagen de contenedor Docker, que se utilizará para la construcción programática del *template* en Cloudformation.

Siguiendo el ejemplo de antes, en la figura 5.5 se puede ver una representación de como quedaría el sistema tras añadir este nuevo componente.

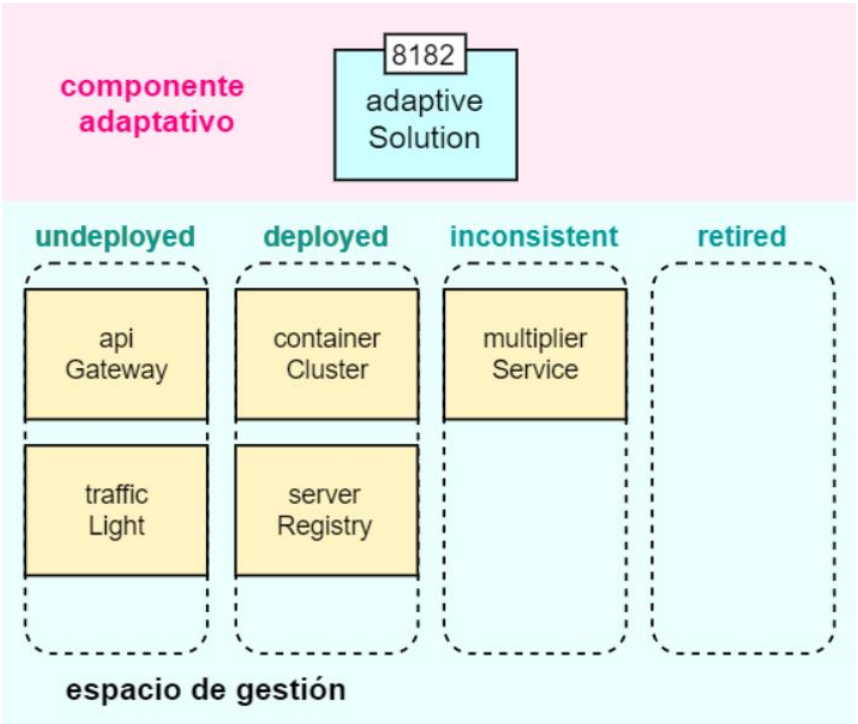


Figura 5.5: Nuevo componente añadido dentro del espacio administrado de adaptiveSolution

Si se realizará una consulta sobre el estado del nuevo recurso devolvería la siguiente respuesta:

```
{
  "id": "trafficLight",
  "status": "undeploy",
  "port": 8761,
  "ip": "public",
  "image": "carizmar/service-registry"
}
```

Es importante destacar el valor del atributo *'template'* ya que representa el proceso realizado durante la configuración e integración del nuevo componente en el sistema. Su valor es el resultado de construir la receta en Cloudformation con los parámetros que se la han proporcionado en el POST como la imagen del contenedor y la configuración de red requerida.

- **PUT (Efector). Cambiar el estado de un componente del sistema administrado**

  Para cambiar el estado actual de un componente es necesario realizar una petición al efector del componente.

```
    efector.localhost:8182/adaptiveSolution/service
    /{nombre del componente}
```

Poniendo como ejemplo el componente que se ha añadido en el paso anterior, se puede cambiar el estado de trafficLight para que se despliegue a partir de su recurso:

```
    efector.localhost:8182/adaptiveSolution/service/trafficLight
```

Con el siguiente cuerpo acompañando la petición:

```
{
  "status": "deploy"
}
```

El atributo *'status'* también acepta como valor las palabras *'undeploy'* para desmontar el servicio de la nube, *'inconsistent'* para indicar la no disponibilidad del servicio o *'retire'* para retirar el servicio y reconfigurarlo si es necesario.

```
{
  "status": "undeploy"
}
```

```
{
  "status": "incosistent"
}
```

```
{
  "status": "retire"
}
```

La solución resultante de las operaciones que se han ejecutado se puede ver representado en la figura 5.6.

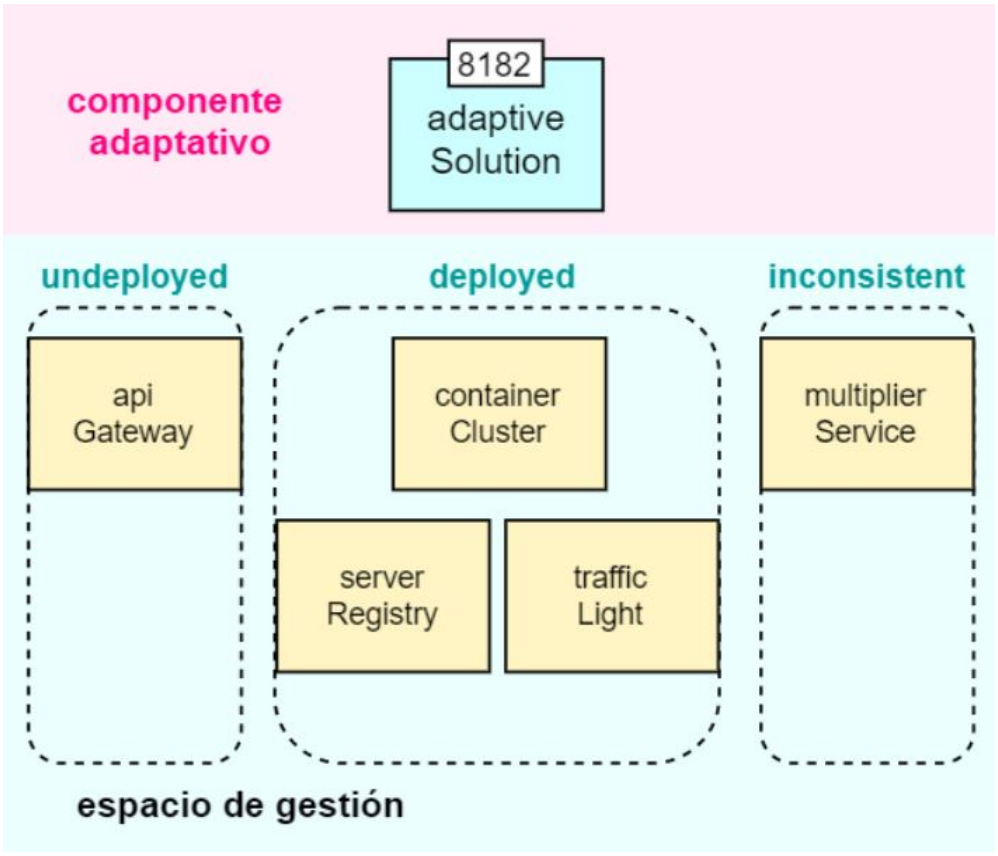


Figura 5.6: Cambio de estado de un componente dentro del espacio administrado de adaptiveSolution

5.4 Implementación de los *templates* de Cloudformation

Cloudformation es una herramienta de infraestructura como código desarrollada por Amazon para aprovisionar infraestructura y recursos virtuales en AWS. El aprovisionamiento es posible utilizando archivos llamados *templates* o recetas que describen y relacionan los recursos necesarios en un lenguaje descriptivo como JSON o YAML.

A través de estas instrucciones es posible el despliegue de recursos virtuales que se agrupan en stacks o pilas durante su ejecución. La agrupación de los recursos permite poder manejar de forma efectiva la dependencia entre ellos y gestionar el conjunto a través de operaciones CRUD.

5.4.1. *Template* para los recursos virtuales

El componente containerCluster del sistema administrado es el encargado de iniciar los recursos virtuales necesarios para que se puedan ejecutar contenedores de aplicaciones en la nube. Por ese motivo, este componente debe concentrar las primeras acciones que se han definido durante el diseño de la solución consistentes en la creación de un *cluster* de contenedores y los recursos de red asociados.

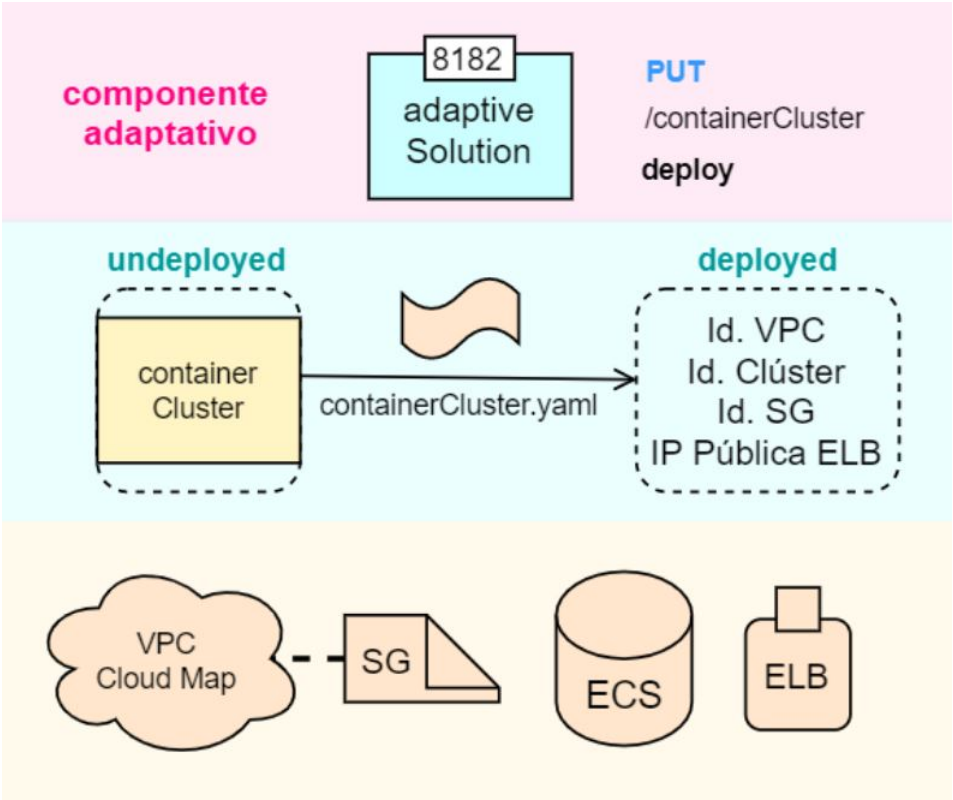


Figura 5.7: Ejecución de un *template* IaC para crear un *cluster* de contenedores y sus recursos de red asociados

Es necesario describir en el *template* que una vez los recursos se han aprovisionado exporten sus identificadores para que el componente adaptativo pueda actualizar el estado de su espacio de gestión. Así mismo, necesita conocer la IP Address (IP) pública desde la que es accesible el *cluster* de contenedores desplegado en ECS.

La implementación y descripción de este *template* puede encontrarse en el apéndice A.

5.4.2. *Template* para el despliegue de microservicios

Como se ha visto en el punto anterior, para que un microservicio pueda ser dado de alta en el sistema administrado es necesario que se le proporcione algunos parámetros de configuración como una imagen de contenedor Docker o las interfaces de red del nodo de computación.

Por este mismo motivo y dado que el componente adaptativo no conoce que microservicios van a desplegarse hasta que no se lo soliciten, no tiene sentido crear una receta específica para los microservicios que se han diseñado. Adicionalmente se podría considerar que la creación determinaría y limitaría la solución.

Es necesario que esta receta se construya programáticamente por el componente adaptativo a través de un *template* genérico que permita construirla personalizando los parámetros de configuración proporcionados a través de la publicación hecha con POST.

Este nuevo *template* genérico para el despliegue de servicios necesita utilizar como parámetros de entrada los identificadores de los recursos virtuales creados en la primera receta, de forma que pueda acceder a los recursos preparados para el despliegue de los microservicios. Una vez la receta se ha construido se iniciará un proceso que cargará de forma automatizada el fichero de instrucciones a S3.

Por otra parte, es necesario controlar el despliegue del contenedor *sidecar* a través del *template* para poder realizar la configuración previa sobre el nodo y después retirarlo para lanzar el contenedor de servicio.

El *sidecar* que se ha utilizado para esta configuración es el que ha sido desarrollado por Docker para el despliegue contenedores en ECS.

```
1 docker pull docker/ecs-searchdomain-sidecar
```

A continuación, se muestra en la figura 5.8 el proceso que lleva a cabo el componente adaptativo para desplegar el microservicio de 'ApiGateway'.

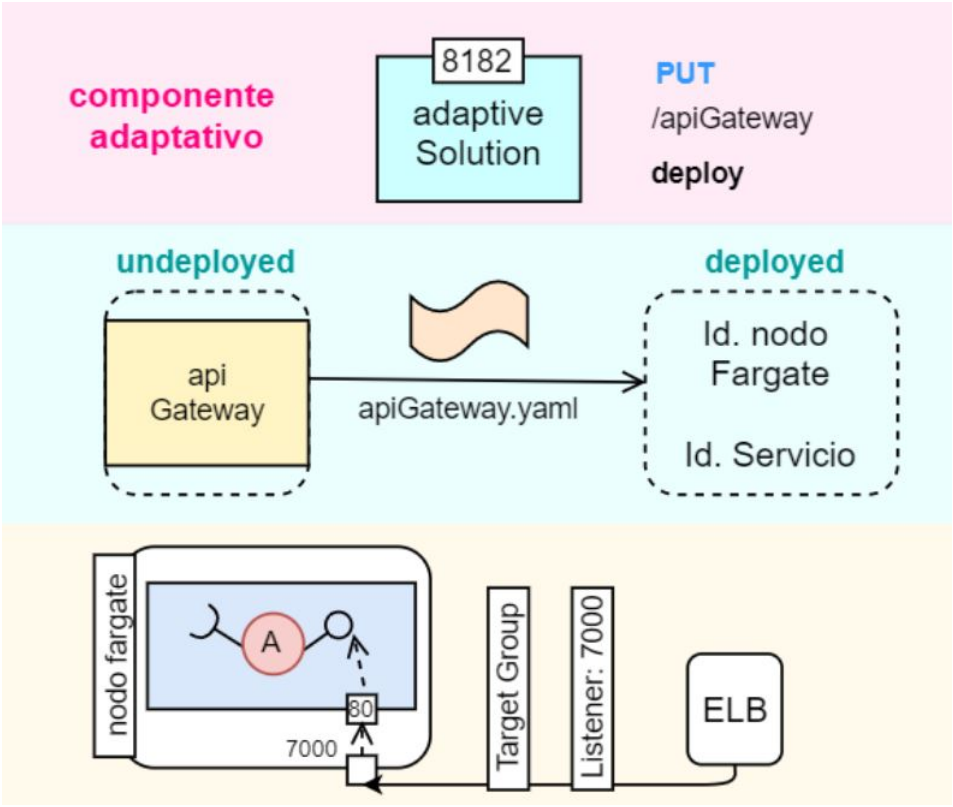


Figura 5.8: Ejecución de un *template* IaC para desplegar un microservicio y sus recursos asociados

El despliegue mostrado en la figura de este microservicio 'ApiGateway' es el que supone la construcción más compleja que debe de realizar el componente adaptativo debido a la multitud de recursos que involucra.

Por tanto, una vez ha sido desplegado el microservicio es necesario publicar su funcionalidad a través del ELB al tratarse de la puerta de enlace para la solución de microservicios.

El listener es el proceso del balanceador de carga que busca las solicitudes que van dirigidas al 'ApiGateway' a través de su puerto 7000. Estas solicitudes una vez encontradas por el listener son redirigidas a un target group que registra todas las instancias redundadas de snodo fargate que implementan el microservicio.

La construcción resultante de este *template* de Cloudformation para el microservicio API Gateway puede encontrarse en el apéndice A.

5.5 Resultados

La puesta en marcha de la solución permite conocer hasta qué punto las tecnologías, herramientas y métodos de desarrollo que se han aplicado durante las fases de diseño e implementación permiten solucionar el problema que se ha expuesto inicialmente a través del análisis de sus resultados.

Su utilización y aplicación de la arquitectura de referencia para el desarrollo de sistemas auto adaptativos que ha sido investigada en profundidad ha demostrado ser perfectamente valida y con el potencial de poder proporcionar capacidades de autogestión a prácticamente cualquier sistema. El resultado muestra la estructuración de la solución en tres tipos de capas distintas que permite no contaminar la implementación base del sistema base del que se requiere dotar de esas capacidades de autoadaptación.

En la primera de estas capas clasificada como sistema administrado, se ha implementado satisfactoriamente una solución genérica que involucra dispositivos físicos con conectividad a Internet lo que garantiza que la implementación puede ser aplicada a cualquier ámbito de la IoT y la Industria 4.0.
Uno de los factores importantes para conseguir esto es haber utilizado una arquitectura de microservicios y patrones de comunicaciones para servicios distribuidos.

Del mismo modo, en la segunda capa, denominada punto de contacto se ha implementado un componente externo al sistema administrado que permite gestionarlo, añadiendo o retirando componentes a demanda y administrando sus recursos que él mismo ha desplegado en infraestructura en la nube. Todas estas operaciones las expone a través de una interfaz de gestión implementada por una API REST para que un tercer componente lo maneje.

La tercera capa que implementa el administrador de autonomía considerando los bucles de control, no se ha desarrollado en profundidad debido a que el desarrollo principal y la idea conceptual se centra en las capas anteriores. De esta forma, en lugar de definir un entorno completo que carezca de funcionalidad real en la práctica, se ha invertido el tiempo en un mejor manejo de la complejidad y accesibilidad de los otros componentes.

Estos bucles de control son los que proporcionan la autonomía y decisión sobre que operaciones de las implementadas en la segunda capa necesita ejecutar para poder adaptarse al entorno. La implementación consigue que la solución sea *adaptive-ready* consiguiendo preparar una adaptación aplicada sobre una infraestructura de nube publica teniendo en cuenta la complejidad de gestión que supone no tener el control completo de los recursos que se están ejecutando.

El resultado final es una solución que permite ejecutarse en entornos cambiantes y está preparada para poder adaptarse, además de tener la capacidad de extenderse sin muchos problemas y ser lo suficientemente genérica como para poder aplicarse en cualquier ámbito.

En el apéndice B puede encontrarse con detalle el proceso que implica la puesta en marcha de la solución de microservicios en la infraestructura en la nube de AWS a través del componente adaptativo.

CAPÍTULO 6
Conclusiones

El planteamiento inicial se basa en la hipótesis de que las soluciones que proponen los principales proveedores de servicios en la nube para los dispositivos IoT son insuficientes. Tras la determinación de los factores e indicadores precisos y el estudio de las plataformas con independencia de la gran variedad de soluciones particulares en el mercado, podemos concluir que a pesar de la funcionalidad no permiten gestionar de manera adecuada sistemas complejos cuando estos tienen unas condiciones de funcionamiento muy cambiantes.

Soluciones que consisten en monitorizar un dispositivo y reiniciarlo cuando se detecta una caída o mantener el estado cuando no está disponible no soluciona el problema y en algunos entornos de producción como los industriales dicha técnica no sería aplicable o podría no resolver el problema completamente. El documento describe que es necesario ir más allá y la necesidad de crear una solución genérica, que no tenga una fuerte dependencia con terceros y plantear la aplicación sobre esta solución de tecnologías de computación autónoma que resuelvan este tipo de problemas.

Una solución que utiliza arquitecturas de microservicios es una aproximación de trabajo más efectiva y eficaz como se ha podido comprobar, ya que permite centrarse en cómo presentar las funcionalidades de los servicios y no tanto en cómo o con que tecnología se implementan. Los diferentes lenguajes de programación o tecnologías utilizadas durante el desarrollo pasan a un segundo plano desde el momento en que es posible meter estos servicios en contenedores con sus propios entornos de ejecución y dependencias.

Lo importante tal como se ha descrito es presentar la información y funcionalidad en un lenguaje que conozca todo el mundo y aplicar los patrones de comunicación que hagan falta para facilitar que estas se produzcan, aunque sean distribuidas. Se ha verificado que la comunicación a través de interfaces API REST o la implementación de colas de mensajes son un mecanismo adecuado para proporcionar estos patrones.

La aplicación de conceptos que provienen de la teoría de control y la computación autónoma en este tipo de sistemas suponen un reto ya que no existe un precedente tecnológico. Sin duda es la parte del proyecto que engloba todo el peso de la investigación y la complejidad de implementación, donde es necesario buscar todo tipo de tecnologías e incluso relacionarlas entre sí para buscar la forma de abordar el problema.

Utilizando como punto de partida 'la arquitectura de referencia para la construcción de sistemas autoadaptables' se permite plantear de qué forma se va a diseñar esta au-

tonomía y establecer una base sobre la que aplicar tecnologías actuales para adaptar y actualizar este modelo que fue planteado hace más de diez años.

Es en este punto es donde se plantea comprobar si la adaptabilidad automatica que define la arquitectura, o dicho de otra forma, la reconfiguración arquitectónica dinámica, es posible aplicarla sobre las infraestructuras elásticas de alta disponibilidad y redundancia que proporcionan los proveedores de servicios en la nube. El resultado demuestra que la capacidad de preparar, adaptar y gestionar de forma autónoma los recursos virtuales proporciona una base para una potencial solución de dispositivos IoT, utilizando herramientas de infraestructura como código de forma dinámica.

6.1 Relación del trabajo desarrollado con estudios cursados

El estudio realizado tenía un destacado objetivo: la conclusión efectiva de un problema que no está resuelto y, por tanto, a pesar de las expectativas se desconocian los resultados de la investigación que se tenía que llevar a cabo.

La gran variabilidad presente y la determinación de las posibles vías de investigación de una forma metódica ha permitido orientar el desarrollo del conjunto de estudio. Como parte de este proceso, a su cierre algunas opciones se han descartado al tratar de analizar la aplicabilidad de una muy amplia cantidad de tecnologías para intentar dar respuesta a los objetivos marcados.

La adquisición de habilidades y conocimientos durante los estudios cursados y los proyectos independientes desarrollados han facilitado, en gran medida, una comprensión mayor de los fundamentos teórico-prácticos elementales para facilitar este proceso de aprendizaje e investigación realizado a lo largo del proyecto.

6.2 Trabajos futuros y ampliaciones

A lo largo de todo el proyecto se ha investigado, diseñado e implementado una solución que aborda el problema que supone el desarrollo de un sistema que debe ser capaz de funcionar en entornos de ejecución complejos donde hay muchos cambios a su alrededor y es necesario poder adaptarse a las circunstancias.

Durante la investigación del estado del arte se han detectado un conjunto muy variado de tecnologías que facilitan la implementación de estos sistemas. Antes de proceder a diseñar la solución ha sido necesario realizar un proceso de selección donde se han valorado las diferentes alternativas tecnológicas y se han priorizado aquellas que permitían simplificar partes de la implementación con el objetivo de poder centrarse en el problema de adaptación.

Estas decisiones llevadas a cabo han limitado el posible potencial que se podría haber conseguido si se hubieran elegido otras alternativas que se han descartado por su complejidad de desarrollo y su alto coste temporal. A continuación, se proponen una serie de ampliaciones que complementan o sustituyen algunos elementos de la solución y que permiten aprovechar algunas ideas que han surgido durante el proyecto.

6.2.1. Implementar el administrador de autonomía y los bucles de control

El resultado de la implementación es una solución *adaptive-ready* que prepara a un sistema administrado para que pueda ser gestionado de forma autónoma todos sus componentes desplegados en infraestructura elástica en la nube.

El planteamiento del proyecto pretendía desde un comienzo, aunque al final no se haya implementado el desarrollar la tercera capa de autonomía consistente en los bucles de control MAPE-K que se han analizado durante la investigación y que proporciona al sistema ya preparado para adaptarse de la capacidad de decidir a hacerlo de acuerdo con las características de su entorno.

La implementación del bucle de control puede realizarse de la misma forma que se ha planteado la de la capa inferior *touchpoint* de sensores y efectores, utilizando una arquitectura de microservicios. Cada uno de los componentes del bucle de control MAPE-K se estructurarían en microservicios redundados y agrupados por monitores, analizadores, planificadores y ejecutores.

Los datos obtenidos y procesados son intercambiados de un servicio a otro utilizando los patrones de comunicación distribuida que se han descrito en el proyecto y los microservicios monitores serían los encargados de comunicarse con los sensores correspondientes de la capa inferior a través de su interfaz de gestión expuesta en una API REST. Por otra, el funcionamiento sería el mismo para los ejecutores con respecto a los efectores. Un diseño conceptual de cómo sería la interacción entre las dos capas de adaptación se puede ver en la figura 6.1

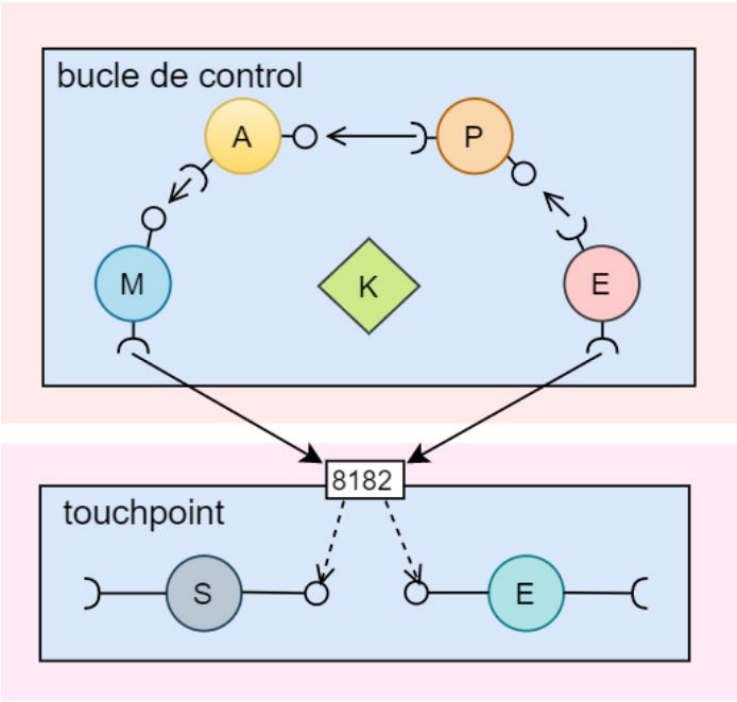


Figura 6.1: Ampliación: Diseño del administrador de autonomía con un bucle de control

6.2.2. Sustituir Amazon ECS por una implementación de Kubernetes en la nube.

Kubernetes hoy en día es una herramienta de referencia en el mercado para la orquestación y despliegue de soluciones de contenedores de aplicaciones distribuidos en *clusters*. Utilizar esta plataforma garantiza una continuidad y un soporte de la comunidad mucho más amplio que la de las alternativas, pero, sin embargo, requiere de tener conocimientos más profundos y detallados sobre su funcionamiento y cómo poner en marcha y configurar una solución de contenedores.

La elección de AWS ECS durante el proyecto viene justificada por la simplicidad de su gestión y la facilidad de puesta en marcha de soluciones de contenedores con una configuración mínima que no es proporcionada por sus competidores. El objetivo del proyecto se centra en analizar el problema de la adaptación autónoma que no requiera un intensivo aprendizaje y permita desacoplar la lógica de la tecnología subyacente es una opción recomendable.

Sin embargo, hay que tener en cuenta que se trata de una solución privativa de un proveedor *cloud* y del cual se depende hasta cierto punto ya que si fuera necesario mover la solución a otra herramienta debería de volver a implementarse los componentes de despliegue automatizados (recursos virtuales).

La mayoría de los proveedores *cloud* han acabado con el tiempo realizando una implementación de Kubernetes como servicio de orquestación en su catálogo de servicios. Este movimiento proporciona una ventaja clave para los desarrolladores que ven como una solución desarrollada para esta herramienta pueda desplegarse sin tener que realizar muchos cambios en la implementación en los diferentes servicios y proveedores *cloud* que la han implementado.

Otro aspecto para tener en cuenta es el movimiento de algunos proveedores *cloud* como Microsoft en el último año, que ha retirado del catálogo de servicios de Azure su implementación de herramienta propia de orquestación de contenedores ACS en favor de su propia implementación de Kubernetes AKS [31].

Nada parece indicar que vaya a suceder lo mismo con la solución de Amazon AWS ECS ya que muchas empresas siguen ejecutando sus aplicaciones y soluciones en el servicio CaaS, aunque sería conveniente migrar a Kubernetes una vez el proyecto ya ha podido validar la prueba de concepto consistente en preparar una plataforma en la nube para ejecutar soluciones autoadaptables.

6.2.3. Utilizar Terraform como herramienta de infraestructura como código para desplegar los microservicios en diferentes *cloud* públicos.

La elección de ECS como servicio orquestador de contenedores viene muy relacionada con la selección de Cloudformation como herramienta de aprovisionamiento de infraestructura como código. Ambas herramientas están desarrolladas por Amazon para su catalogo de servicios en la nube y por lo tanto las funcionalidades que es posible conseguir cuando Cloudformation se describen los recursos virtuales de AWS, no es posible alcanzarlas utilizando otras herramientas.

Terraform es una herramienta alternativa open-source para el aprovisionamiento de recursos virtuales mediante el uso de recetas implementadas en su propio lenguaje llamado HCL. El punto fuerte de esta herramienta es que posibilita gestionar recursos de los principales proveedores de *cloud* públicos, además de infraestructura de nubes privadas.

El diseño de la capa de adaptación de sensores y efectores se ha realizado pensando en esta posibilidad y pudiendo implementarla sin realizar ningún esfuerzo gracias al espacio administrado que se ha definido.

Este espacio administrado o de gestión proporciona una estructuración de componentes que puede ser tanto microservicios como recursos virtuales de proveedores *cloud* distintos. La figura 6.2 representa esta posibilidad donde uno de los recursos podría ser un *cluster* de contenedores en AWS desplegado mediante una receta de Cloudformation y el otro podría ser un *cluster* de contenedores en Azure desplegado mediante una receta en Terraform.

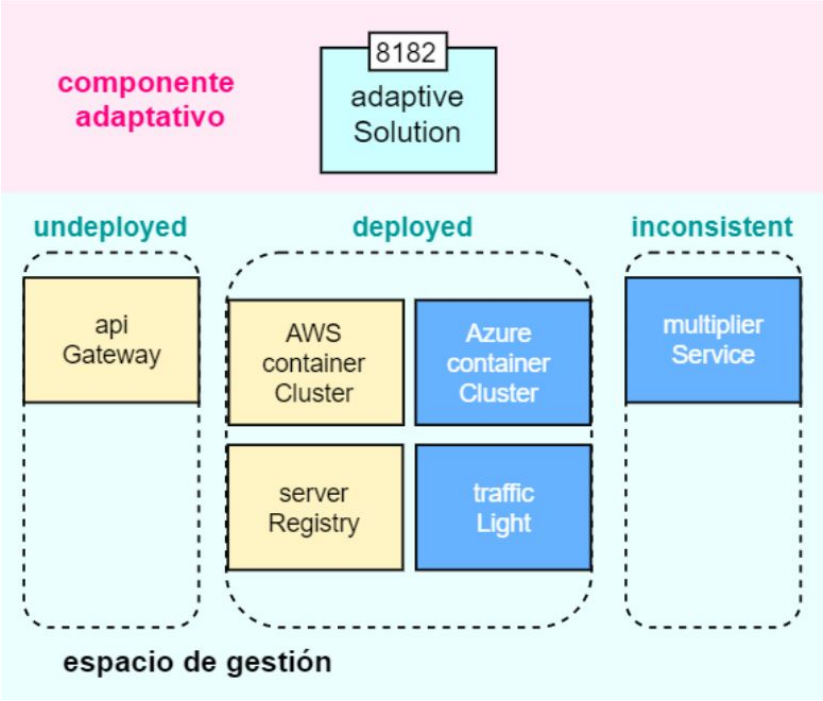


Figura 6.2: Ampliación: Multiples recursos de proveedores *cloud* dentro del espacio gestionado

Los cambios que harían falta implementar implicarían volver a diseñar la construcción de recetas y los parámetros de configuración necesarios para añadir un nuevo componente al sistema, donde ahora sería necesario especificar que proveedor de recursos virtuales se debería de utilizar para desplegar el microservicio.

Esta ampliación podría incluso plantear el hecho de construir los mecanismos y recursos de red necesarios para que los microservicios desplegados en diferentes *clusters* e infraestructuras de proveedores *cloud* pudieran comunicarse de manera privada sin saber en que plataforma se están ejecutando.

Siglario

ACS Azure Container Service. 20, 84

AKS Azure Kubernetes Service. 21, 36, 84

API Interfaz de programación de aplicaciones. 15, 19, 35, 42–44, 46, 49, 50, 53, 56–59, 61, 62, 65, 68, 70, 78, 79, 81, 83

AWS Amazon Web Services. 13, 20, 21, 26, 36, 50, 53, 57, 58, 70, 76, 84, 85

CaaS Container as a Service. 19–21, 27, 35, 84

CoAP Constrained Application Protocol. 1

CRUD Create, Read, Update and Delete. 26, 76

ECS Elastic Container Service. 20, 36, 37, 53, 57, 58, 70, 77, 84

EKS Elastic Kubernetes Service. 21, 36, 37

ELB Elastic Load Balancer. 50, 72, 73, 78

GCP Google Cloud Platform. 20, 26

GKE Google Kubernetes Engine. 21, 36

IaaS Infraestructure as a Service. 19, 20

IaC Infrastructure as Code. 26, 57

IoT Internet of Things. 1, 2, 6, 13–17, 19, 27, 29, 46, 79, 82

IP IP Address. 77

LPWA Low Power Wide Area. 1

MQTT Message Queuing Telemetry Transport. 1, 14, 16, 62

PaaS Platform as a Service. 13, 19

REST Representational state transfer. 16, 46, 55, 56, 58–60, 62, 65, 68, 70, 79, 81, 83

S3 Amazon Simple Storage Service. 72, 77

SDK Software Development Kit. 57, 68

UML Unified Modeling Language. 39

VPC Virtual Private Cloud. 36, 50, 57

Bibliografía

[1] Ray, P. P. (2018, July 1). A survey on Internet of Things architectures. Journal of King Saud University - Computer and Information Sciences. King Saud bin Abdulaziz University. https://doi.org/10.1016/j.jksuci.2016.10.003

[2] RFC 8376 (2018). Low-Power Wide Area Network (LPWAN) Overview. https://datatracker.ietf.org/doc/html/rfc8376

[3] Villamil, S., Hernández, C., Tarazona, G. (2020). An overview of internet of things. Telkomnika (Telecommunication Computing Electronics and Control), 18(5), 2320–2327. https://doi.org/10.12928/TELKOMNIKA.v18i5.15911

[4] Asplund, M., Nadjm-Tehrani, S. (2016). Attitudes and Perceptions of IoT Security in Critical Societal Services. IEEE Access, 4, 2130–2138. https://doi.org/10.1109/ACCESS.2016.2560919

[5] Lu, Y. (2017, June 1). Industry 4.0: A survey on technologies, applications and open research issues. Journal of Industrial Information Integration. Elsevier B.V. https://doi.org/10.1016/j.jii.2017.04.005

[6] Canedo, A. (2016). Industrial IoT lifecycle via digital twins. In 2016 International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2016. Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1145/2968456.2974007

[7] Opara-Martins, J., Sahandi, R., Tian, F. (2016). Critical analysis of vendor lock-in and its impact on cloud computing migration: a business perspective. Journal of Cloud Computing, 5(1). https://doi.org/10.1186/s13677-016-0054-z

[8] De, S., Barnaghi, P., Bauer, M., Meissner, S. (2011). Service modelling for the Internet of Things. In 2011 Federated Conference on Computer Science and Information Systems, FedCSIS 2011 (pp. 949–955).

[9] Vresk, T., Cavrak, I. (2016). Architecture of an interoperable IoT platform based on microservices. In 2016 39th International Convention on Information and Communication Technology, Electronics and Microelectronics, MIPRO 2016 - Proceedings (pp. 1196–1201). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/MIPRO.2016.7522321

[10] Trilles, S., González-Pérez, A., Huerta, J. (2020). An IoT platform based on microservices and serverless paradigms for smart farming purposes. Sensors (Switzerland), 20(8). https://doi.org/10.3390/s20082418

[11] Razzaque, M. A., Milojevic-Jevric, M., Palade, A., Cla, S. (2016). Middleware for internet of things: A survey. IEEE Internet of Things Journal, 3(1), 70–95. https://doi.org/10.1109/JIOT.2015.2498900

[12] Palade, A., Cabrera, C., White, G., Razzaque, M. A., Clarke, S. (2017). Middleware for Internet of Things: A quantitative evaluation in small scale. In 18th IEEE International Symposium on A World of Wireless, Mobile and Multimedia Networks, WoWMoM 2017 - Conference. Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/WoWMoM.2017.7974340

[13] Thramboulidis, K., Vachtsevanou, D. C., Solanos, A. (2018). Cyber-physical microservices: An IoT-based framework for manufacturing systems. In Proceedings - 2018 IEEE Industrial Cyber-Physical Systems, ICPS 2018 (pp. 232–239). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/ICPHYS.2018.8387665

[14] Ciavotta, M., Alge, M., Menato, S., Rovere, D., Pedrazzoli, P. (2017). A Microservice-based Middleware for the Digital Factory. Procedia Manufacturing, 11, 931–938. https://doi.org/10.1016/j.promfg.2017.07.197

[15] Innerbichler, J., Gonul, S., Damjanovic-Behrendt, V., Mandler, B., Strohmeier, F. (2017). NIMBLE collaborative platform: Microservice architectural approach to federated IoT. In GIoTS 2017 - Global Internet of Things Summit, Proceedings. Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/GIOTS.2017.8016216

[16] Ciuffoletti, A. (2015). Automated Deployment of a Microservice-based Monitoring Infrastructure. In Procedia Computer Science (Vol. 68, pp. 163–172). Elsevier B.V. https://doi.org/10.1016/j.procs.2015.09.232

[17] Morabito, R., Kjällman, J., Komu, M. (2015). Hypervisors vs. lightweight virtualization: A performance comparison. In Proceedings - 2015 IEEE International Conference on Cloud Engineering, IC2E 2015 (pp. 386–393). Institute of Electrical and Electronics Engineers Inc. https://doi.org/10.1109/IC2E.2015.74

[18] De Prado, R. P., García-Galán, S., Muñoz-Expósito, J. E., Marchewka, A., Ruiz-Reyes, N. (2020). Smart containers schedulers for microservices provision in cloud-fog-IoT networks. challenges and opportunities. Sensors (Switzerland), 20(6). https://doi.org/10.3390/s20061714

[19] Kiss, T., Kacsuk, P., Kovacs, J., Rakoczi, B., Hajnal, A., Farkas, A., ... Terstyanszky, G. (2019). MiCADO—Microservice-based Cloud Application-level Dynamic Orchestrator. Future Generation Computer Systems, 94, 937–946. https://doi.org/10.1016/j.future.2017.09.050

[20] Hussein, M. K., Mousa, M. H., Alqarni, M. A. (2019). A placement architecture for a container as a service (CaaS) in a cloud environment. Journal of Cloud Computing, 8(1). https://doi.org/10.1186/s13677-019-0131-1

[21] Pahl, C. (2015). Containerization and the PaaS Cloud. IEEE Cloud Computing, 2(3), 24–31. https://doi.org/10.1109/MCC.2015.51

[22] Mcluckie, C. (2014). Containers, VMs, Kubernetes and VMware. Google Cloud Platform Blog. https://cloudplatform.googleblog.com/2014/08/containers-vms-kubernetes-and-vmware.html

[23] Bernstein, D. (2014). Containers and cloud: From LXC to docker to kubernetes. IEEE Cloud Computing, 1(3), 81–84. https://doi.org/10.1109/MCC.2014.51

[24] Cloud Native Computing Foundation. What is Kubernetes - Kubernetes. Kubernetes.Io (2019).

[25] Kephart, J. O., Chess, D. M. (2003). The vision of autonomic computing. Computer, 36(1), 41–50. https://doi.org/10.1109/MC.2003.1160055

[26] IBM. (2006). An architectural blueprint for autonomic computing. IBM White Paper, http://scholar.google.com/scholar?hl=en&btnG=Search&q=intitle:An+architectural+blueprint+for+autonomic+computing+.#0

[27] Ivanov, D., Sethi, S., Dolgui, A., Sokolov, B. (2018, January 1). A survey on control theory applications to operational systems, supply chain management, and Industry 4.0. Annual Reviews in Control. Elsevier Ltd. https://doi.org/10.1016/j.arcontrol.2018.10.014

[28] Brun, Y., Di Marzo Serugendo, G., Gacek, C., Giese, H., Kienle, H., Litoiu, M., Shaw, M. (2009). Engineering self-adaptive systems through feedback loops. In Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) (Vol. 5525 LNCS, pp. 48–70). https://doi.org/10.1007/978-3-642-02161-9_3

[29] Yamato, Y., Muroi, M., Tanaka, K., Uchimura, M. (2014). Development of template management technology for easy deployment of virtual resources on OpenStack. Journal of Cloud Computing, 3(1), 1–12. https://doi.org/10.1186/s13677-014-0007-3

[30] Kramer, J., Magee, J. (2007). Self-managed systems: An architectural challenge. In FoSE 2007: Future of Software Engineering (pp. 259–268). https://doi.org/10.1109/FOSE.2007.19

[31] Microsoft Azure Documentation. https://azure.microsoft.com/es-es/updates/azure-container-service-will-retire-on-january-31-2020/

APÉNDICE A
Templates de Cloudformation

El apéndice A muestra un ejemplo de la construcción que realiza el componente adaptativo de los *templates* de Cloudformation a través de los parámetros de entrada configurados en el espacio administrado.

Se estructura de la siguiente forma:

- **A1.** Aprovisionamiento del *cluster* de contenedores de Amazon ECS y la configuración de los recursos de red asociados al VPC.

- **A2.** Despliegue del contenedor de microservicios Api Gateway. Esta construcción dinámica del *template* es la más compleja y completa al involucrar a todos los recursos de red creados en la anterior aprovisionamiento, además de crear nuevos.

En los dos *templates* se pueden encontrar comentarios en color rojo detallando las descripciones más relevantes durante la implementación.

A.1 Aprovisionamiento de un *cluster* de contenedores y sus recursos de red asociados

```
{
AWSTemplateFormatVersion: 2010-09-09
Resources:
  #Descripcion del Balanceador de carga elastico (ELB)
    #Asociacion con diferentes zonas de disponibilidad
  LoadBalancer:
    Properties:
      Scheme: internet-facing
      Subnets:
      - subnet-62a3dd3d
      - subnet-9734b5a6
      - subnet-0793ee61
      - subnet-2cefd822
      - subnet-092a2744
      - subnet-ccc6beed
      Tags:
```

```
17        - Key: com.carizmar.solution.project
18          Value: msdeploy
19        Type: network
20      Type: AWS::ElasticLoadBalancingV2::LoadBalancer
21
22    #Descripcion del espacio de nombres privados en CloudMap
23    CloudMap:
24      Properties:
25        Description: Espacio de nombres para los nodos
26        Name: msdeploy.local
27        Vpc: vpc-4bd85136
28      Type: AWS::ServiceDiscovery::PrivateDnsNamespace
29
30    #Descripcion de un nuevo clutser de contenedores en ECS
31    Cluster:
32      Properties:
33        ClusterName: msdeploy
34        Tags:
35        - Key: com.carizmar.solution.project
36          Value: msdeploy
37      Type: AWS::ECS::Cluster
38
39    #Descripcion de un "grupo de seguridad" asociado al cluster de
          contenedores
40    ContainerClusterNetwork:
41      Properties:
42        GroupDescription: Grupo de seguridad para la red del
            cluster
43        Tags:
44        - Key: com.carizmar.solution.project
45          Value: msdeploy
46        - Key: com.carizmar.solution.red
47          Value: msdeploy_dockernet-network
48        VpcId: vpc-4bd85136
49      Type: AWS::EC2::SecurityGroup
50
51    #Permitir la comunicacion dentro del grupo de seguridad
52    ContainerClusterNetworkIngress:
53      Properties:
54        Description: Permitir comunicaciones en la red del cluster
55        GroupId:
56          Ref: ContainerClusterNetwork
57        IpProtocol: "-1"
58        SourceSecurityGroupId:
59          Ref: ContainerClusterNetwork
60      Type: AWS::EC2::SecurityGroupIngress
61
62  #Exportar parametros para utilizarlos en otras recetas
63  Outputs:
64    #Exportar IP Publica del ELB
65    LoadBalancerDNSName:
66      Description: "ELB: Nombre DNS"
```

```
67      Value: !GetAtt LoadBalancer.DNSName
68      Export:
69        Name: !Sub "${AWS::StackName}-DNS"
70    #Exportar ID del cluster
71    ClusterId:
72      Description: "ECS: Cluster ID"
73      Value: !Ref Cluster
74      Export:
75        Name: !Sub "${AWS::StackName}-CLUSTERID"
76    #Exportar ID del "grupo de seguridad" del cluster
77    SecurityGroupRef:
78      Description: "SG: Cluster"
79      Value: !Ref ContainerClusterNetwork
80      Export:
81        Name: !Sub "${AWS::StackName}-SGREF"
82    #Exportar ID del espacio de nombres privado en CloudMap
83    CloudMapRef:
84      Description: "CloudMap: Namespace ID"
85      Value: !Ref CloudMap
86      Export:
87        Name: !Sub "${AWS::StackName}-CLMID"
```

A.2 Despliegue de un contenedor de microservicios

```
AWSTemplateFormatVersion: 2010-09-09
Parameters:
  #Se pasa como entrada la referencia del stack que ha
     aprovisionado el cluster ecs y los recursos de red
  ReferenceStackName:
    Description: Referencia a la pila que contiene el cluster de
       contenedores
    Type: String
    MinLength: 1
    MaxLength: 255
    AllowedPattern: '^[a-zA-Z][-a-zA-Z0-9]*$'
    Default: cluster-tfm
Resources:

  #Descripcion de la tarea (nodo) AWS Fargate en ECS
  NodoFargateTaskDefinition:
    Type: AWS::ECS::TaskDefinition
    Properties:
      Family: msdeploy-apigateway-service
      ContainerDefinitions:
      #Ejecutar el contenedor sidecar para configurar el
         namespace privado en el nodo
      - Command:
        - us-east-1.compute.internal
        - msdeploy.local
        Essential: false
        Image: docker/ecs-searchdomain-sidecar:1.0
        Name: ApiGateway_Config_InitContainer
      - DependsOn:
        - Condition: SUCCESS
          ContainerName: ApiGateway_Config_InitContainer
        Essential: true
        #Definicion del contenedor que ejecuta el microservicio
           principal
        Image: carizmar/api-gateway
        Name: 'apigateway-service'
        #Enlace entre los puertos expuestos del nodo y el
           contenedor
        PortMappings:
        - ContainerPort: 80
          HostPort: 7000
          Protocol: tcp
      #Requisitos cpu, memoria y red del nodo
      NetworkMode: awsvpc
      Cpu: "256"
      Memory: "512"
      RequiresCompatibilities:
      - FARGATE

  #Descripci n de un nuevo Target Group (7000/tcp) en el ELB
```

```
46    ApiGatewayTCP7000TargetGroup:
47      Properties:
48        Port: 7000
49        Protocol: TCP
50        Tags:
51        - Key: com.carizmar.solution.project
52          Value: msdeploy
53        TargetType: ip
54        VpcId: vpc-4bd85136
55      Type: AWS::ElasticLoadBalancingV2::TargetGroup
56
57    #Descripcion de un nuevo Listener (7000/tcp) en el ELB
58      #Funcionamiento: Redirigir el trafico al Target Group (7000/
          tcp)
59    ApiGatewayTCP7000Listener:
60      Properties:
61        DefaultActions:
62        - ForwardConfig:
63            TargetGroups:
64            - TargetGroupArn:
65                Ref: ApiGatewayTCP7000TargetGroup
66          Type: forward
67        LoadBalancerArn:
68          Ref: LoadBalancer
69        Port: 7000
70        Protocol: TCP
71      Type: AWS::ElasticLoadBalancingV2::Listener
72
73    #A adir entrada del puerto  7000 en el "grupo de seguridad"
        del cluster de contenedores
74    ClusterNetwork7000Ingress:
75      Properties:
76        CidrIp: 0.0.0.0/0
77        Description: Ingreso de la entrada 7000/tcp en el sg del
            cluster
78        FromPort: 7000
79        GroupId:
80          Ref: ContainerClusterNetwork
81        IpProtocol: TCP
82        ToPort: 7000
83      Type: AWS::EC2::SecurityGroupIngress
84
85    #Se crea una entrada en el namespace privado de CloudMap
86    ServiceDiscoveryEntry:
87      Properties:
88        Description: Entrada en el namespace privado
89        DnsConfig:
90          DnsRecords:
91          - TTL: 60
92            Type: A
93          RoutingPolicy: MULTIVALUE
94        HealthCheckCustomConfig:
```

```
95          FailureThreshold: 1
96        Name: apigateway-service
97        NamespaceId:
98          Fn::ImportValue:
99            !Sub "${ReferenceStackName}-CLMID"
100     Type: AWS::ServiceDiscovery::Service
101
102   #Descripcion del servicio
103   ApiGatewayService:
104     Type: AWS::ECS::Service
105     Properties:
106       #Se asigna el cluster importando su valor del stack pasado
            como entrada
107       Cluster:
108         Fn::ImportValue:
109           !Sub "${ReferenceStackName}-CLUSTERID"
110       #Numero de instancias del mismo servicio
111       DesiredCount: 1
112       LoadBalancers:
113       - ContainerName: apigateway-service
114         ContainerPort: 7000
115         #Se registra el servicio dentro del target
116         TargetGroupArn:
117           Ref: ApiGatewayTCP7000TargetGroup
118       LaunchType: FARGATE
119       ServiceName: api-gateway
120       #Configuracion de red
121       NetworkConfiguration:
122         AwsvpcConfiguration:
123           #Asignacion de ip publica o privada
124           AssignPublicIp: ENABLED
125           #Se importa el grupo de seguridad del cluster
126           SecurityGroups:
127           - Fn::ImportValue: !Sub "${ReferenceStackName}-SGREF"
128           #Se asigna una de las siguientes zonas de
                disponibilidad
129           Subnets:
130           - subnet-62a3dd3d
131           - subnet-9734b5a6
132           - subnet-0793ee61
133           - subnet-2cefd822
134           - subnet-092a2744
135           - subnet-ccc6beed
136       #Se registra dentro de la entrada creada en CloudMap
137       ServiceRegistries:
138       - RegistryArn:
139           Fn::GetAtt:
140           - ServiceDiscoveryEntry
141           - Arn
142       #Se asigna tarea (nodo) de ejecucion creado de AWS Fargate
143       TaskDefinition:
144         Ref: NodoFargateTaskDefinition
```

APÉNDICE B
Puesta en marcha de la solución

- **GET. Conocer el estado inicial del sistema administrado**

```
sensor.touchpoint:8182/adaptiveSolution
```

Respuesta:

```
{
    "undeployed": [
        {
            "containerCluster": 0
        }
    ],
    "deployed": [],
    "inconsistent": [],
    "retired": [],
    "id": "touchpoint"
}
```

- **GET. Consultar la configuración de despliegue del *cluster* de ECS**

```
sensor.touchpoint:8182/adaptiveSolution/service/containerCluster
```

Respuesta:

```
{
    "template": "https://carizmar-tfm.s3.amazonaws.com/containerCluster.
        yml",
    "port": 0,
    "ip": "aws",
    "id": "containerCluster",
    "status": "UNDEPLOYED"
}
```

- **PUT. Desplegar el *cluster* de contenedores y los recursos de red asociados**

```
efector.touchpoint:8182/adaptiveSolution/service/containerCluster
```

Cuerpo de la petición:

```
{
    "status":"deploy"
}
```

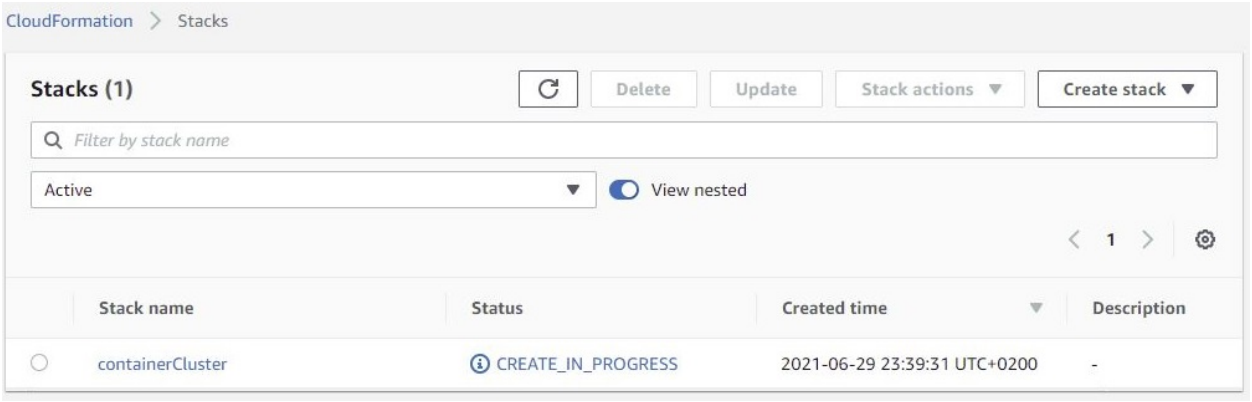


Figura B.1: Cloudformation: Aprovisionamiento en proceso del *cluster* y los recursos de red

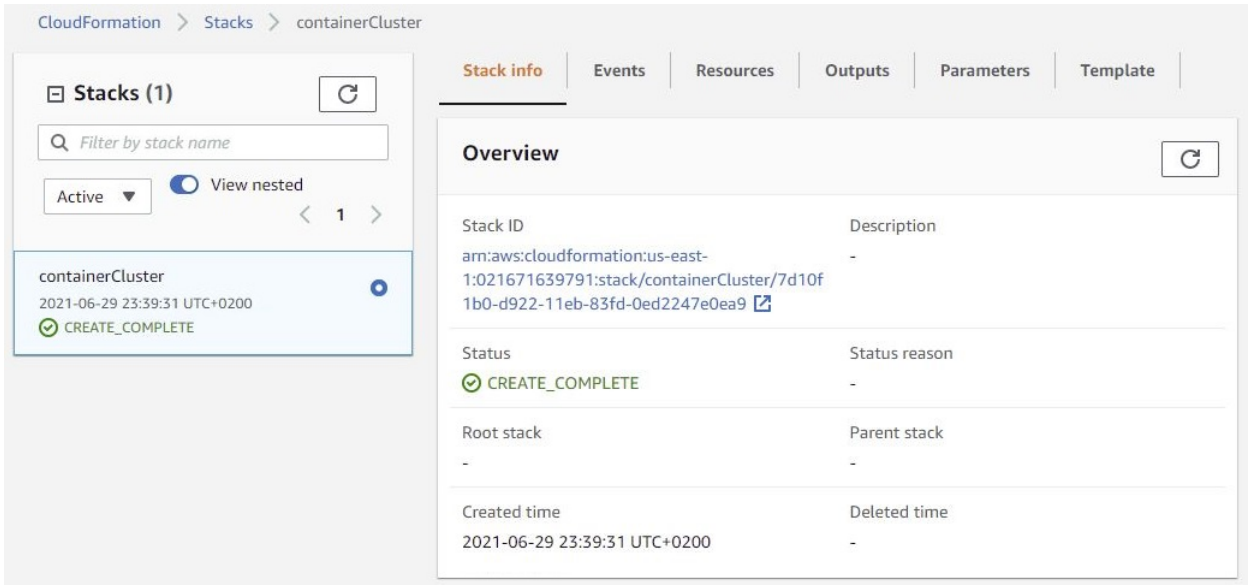


Figura B.2: Cloudformation: Aprovisionamiento completado del *cluster* y los recursos de red

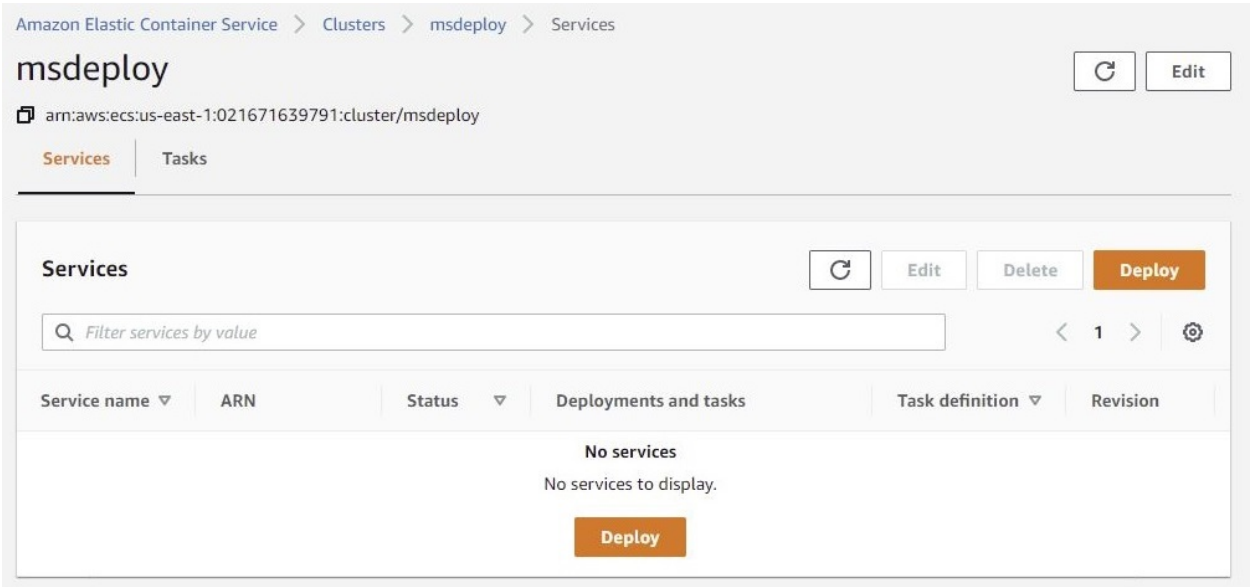


Figura B.3: ECS: *cluster* de contenedores msdeploy desplegado

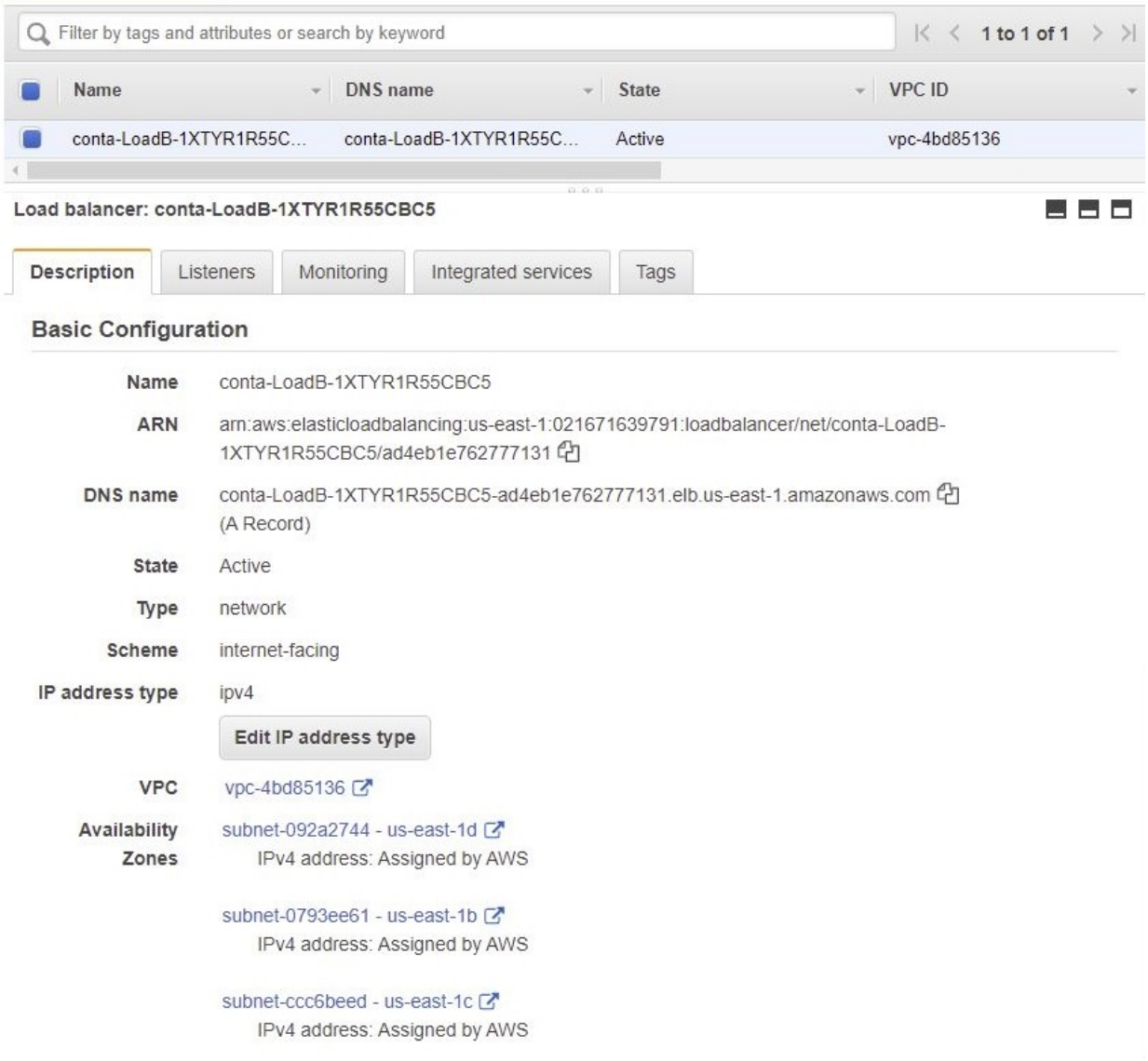


Figura B.4: ELB: Balanceador de carga desplegado y configurado

- **GET. Consultar la configuración desplegada del *cluster* de ECS**

```
sensor.touchpoint:8182/adaptiveSolution/service/containerCluster
```

Respuesta:

```
{
    "template": "https://carizmar-tfm.s3.amazonaws.com/containerCluster.
        yml",
    "port": 0,
    "ip": "conta-LoadB-1XTYR1R55CBC5-ad4eb1e762777131.elb.us-east-1.
        amazonaws.com",
    "id": "containerCluster",
    "status": "DEPLOYED"
}
```

- **POST. Añadir el microservicio *Service Registry* al espacio administrado**

```
efector.touchpoint:8182/adaptiveSolution
```

Cuerpo de la petición:

```
{
  "id": "serverRegistry",
  "status": "undeploy",
  "port": 8761,
  "ip": "public",
  "image": "carizmar/service-registry"
}
```

- **GET. Consultar la configuración del microservicio serverRegistry**

```
    sensor.touchpoint:8182/adaptiveSolution/service/serverRegistry
```

Respuesta:

```
{
    "template": "https://carizmar-tfm.s3.amazonaws.com/serverRegistry.yml
        ",
    "port": 8761,
    "ip": "public",
    "id": "serverRegistry",
    "status": "UNDEPLOYED"
}
```

- **PUT. Desplegar el microservicio serverRegistry en el *cluster***

```
    efector.touchpoint:8182/adaptiveSolution/service/serverRegistry
```

Cuerpo de la petición:

```
{
    "status":"deploy"
}
```

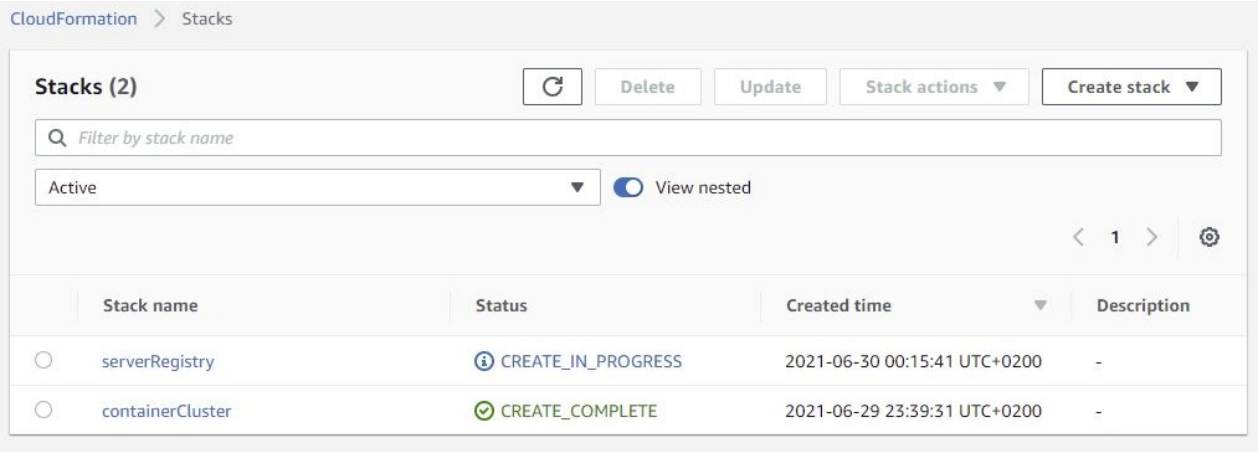


Figura B.5: Cloudformation: Despliegue en proceso de la receta serverRegistry

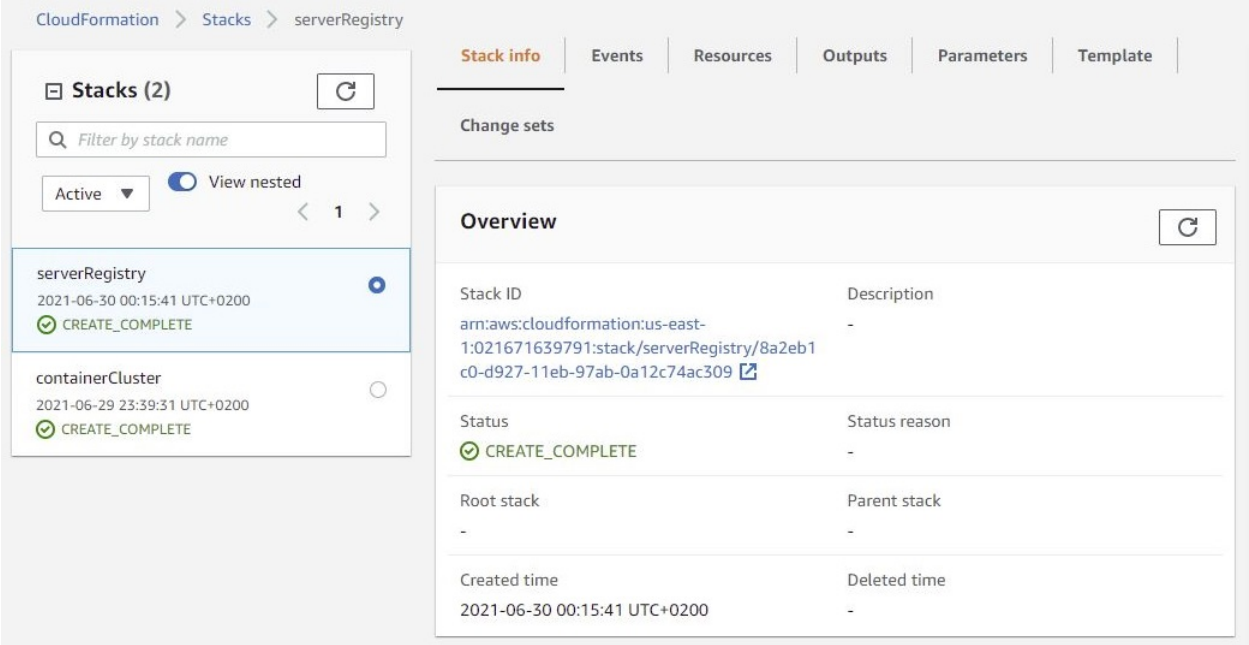


Figura B.6: Cloudformation: Despliegue completado de la receta serverRegistry

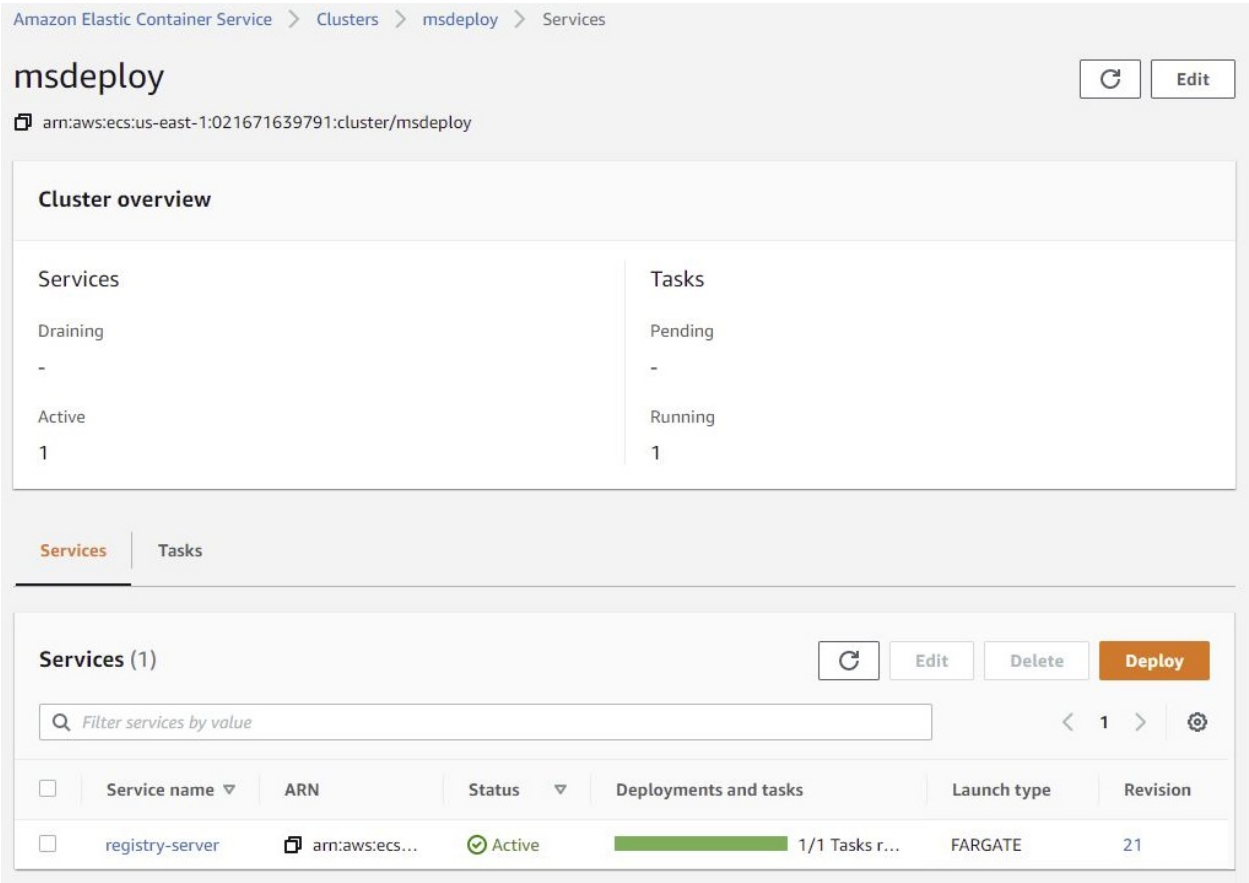


Figura B.7: ECS: Microservicio serverRegistry desplegado en el *cluster*

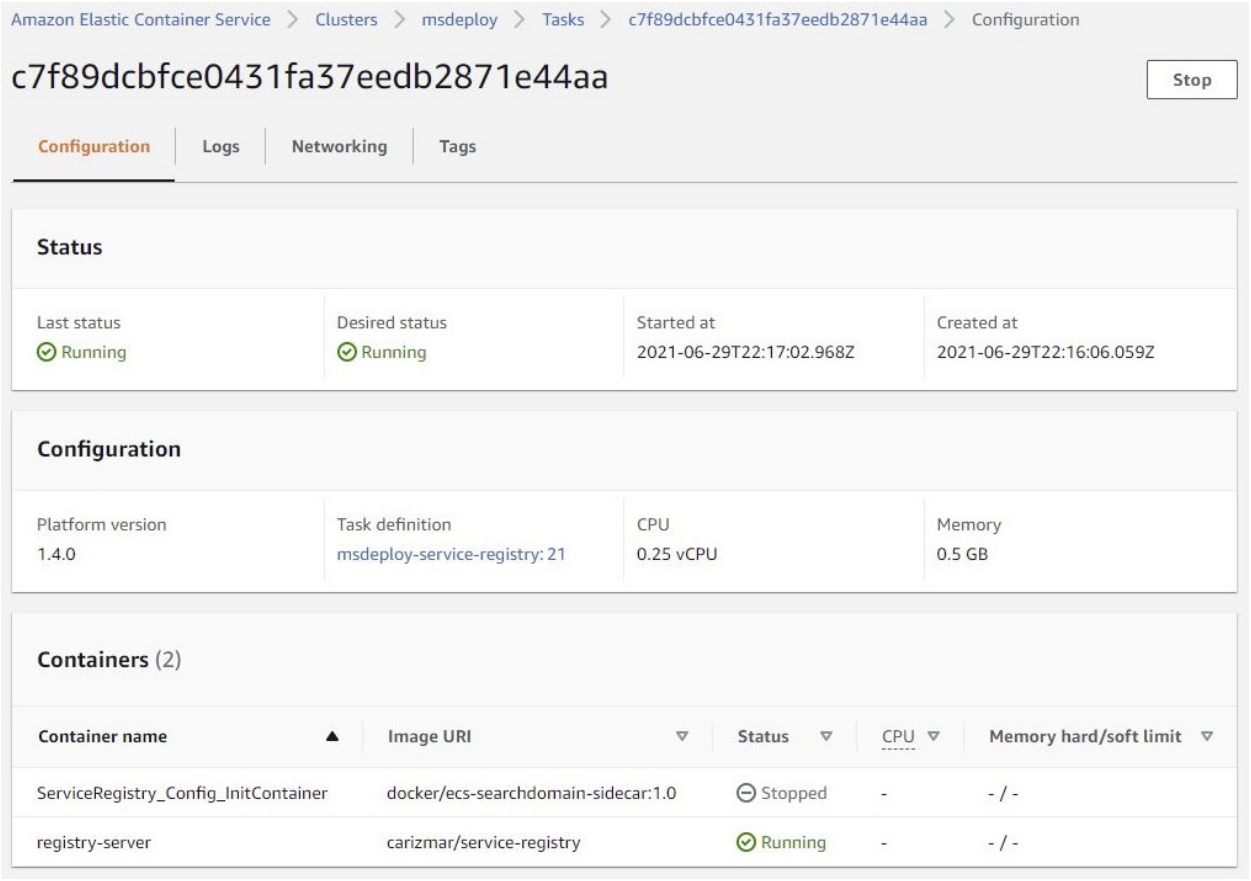


Figura B.8: Fargate: Ejecución de los contenedores *sidecar* y service-registry en la tarea (nodo)

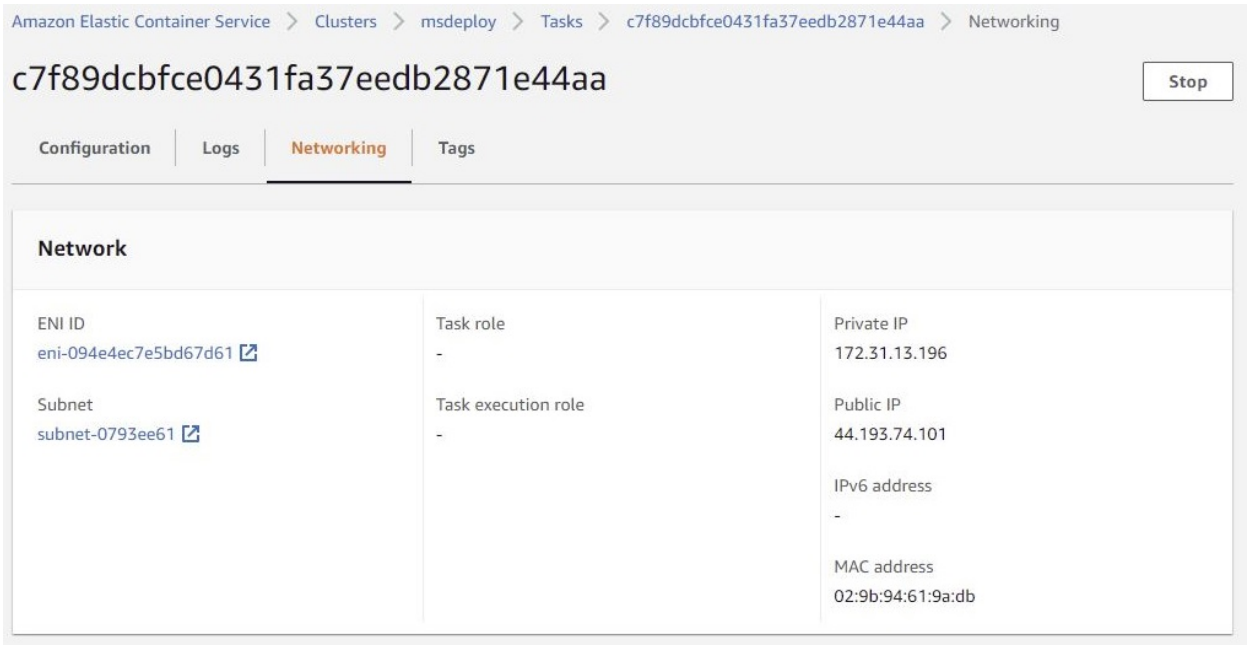


Figura B.9: Fargate: Configuración de red de la tarea (nodo) del ServiceRegistry

- **GET. Consultar la configuración desplegada del microservicio serverRegistry**

```
sensor.touchpoint:8182/adaptiveSolution/service/serverRegistry
```

Respuesta:

```
{
    "template": "https://carizmar-tfm.s3.amazonaws.com/serverRegistry.yml
        ",
    "port": 8761,
    "ip": "conta-LoadB-1XTYR1R55CBC5-ad4eb1e762777131.elb.us-east-1.
        amazonaws.com",
    "id": "serverRegistry",
    "status": "DEPLOYED"
}
```

- **POST. Añadir el microservicio apiGateway al espacio administrado**

```
efector.touchpoint:8182/adaptiveSolution
```

Cuerpo de la petición:

```
{
  "id": "apiGateway",
  "status": "undeploy",
  "port": 7000,
  "ip": "public",
  "image": "carizmar/api-gateway"
}
```

- **GET. Consultar la configuración del microservicio apiGateway**

```
sensor.touchpoint:8182/adaptiveSolution/service/apiGateway
```

Respuesta:

```
{
    "template": "https://carizmar-tfm.s3.amazonaws.com/apiGateway.yml",
    "port": 7000,
    "ip": "public",
    "id": "apiGateway",
    "status": "UNDEPLOYED"
}
```

- **PUT. Desplegar el microservicio apiGateway en el *cluster***

```
efector.touchpoint:8182/adaptiveSolution/service/apiGateway
```

Cuerpo de la petición:

```
{
    "status":"deploy"
}
```

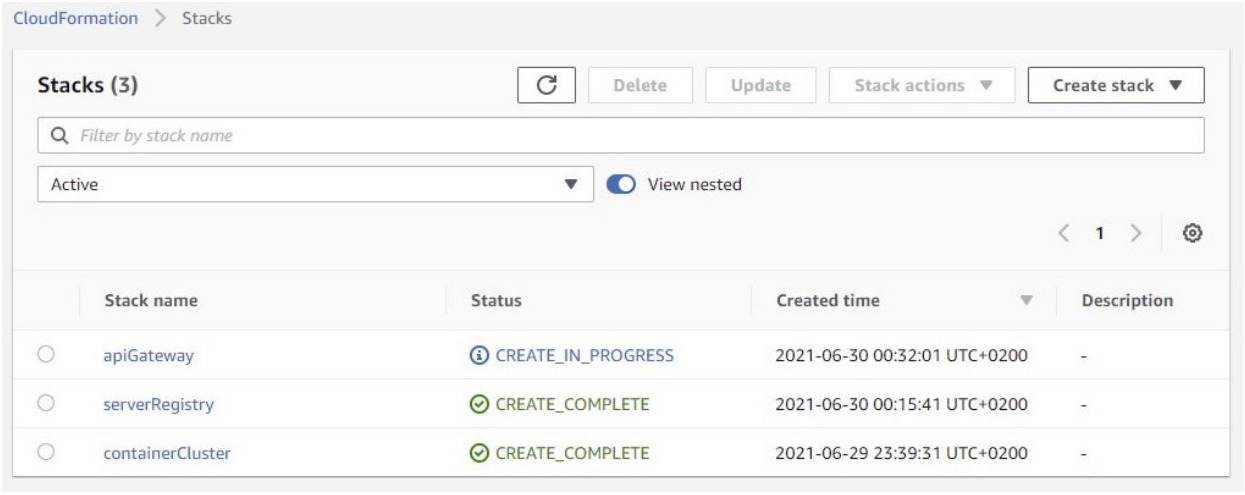


Figura B.10: Cloudformation: Despliegue de la receta ApiGateway

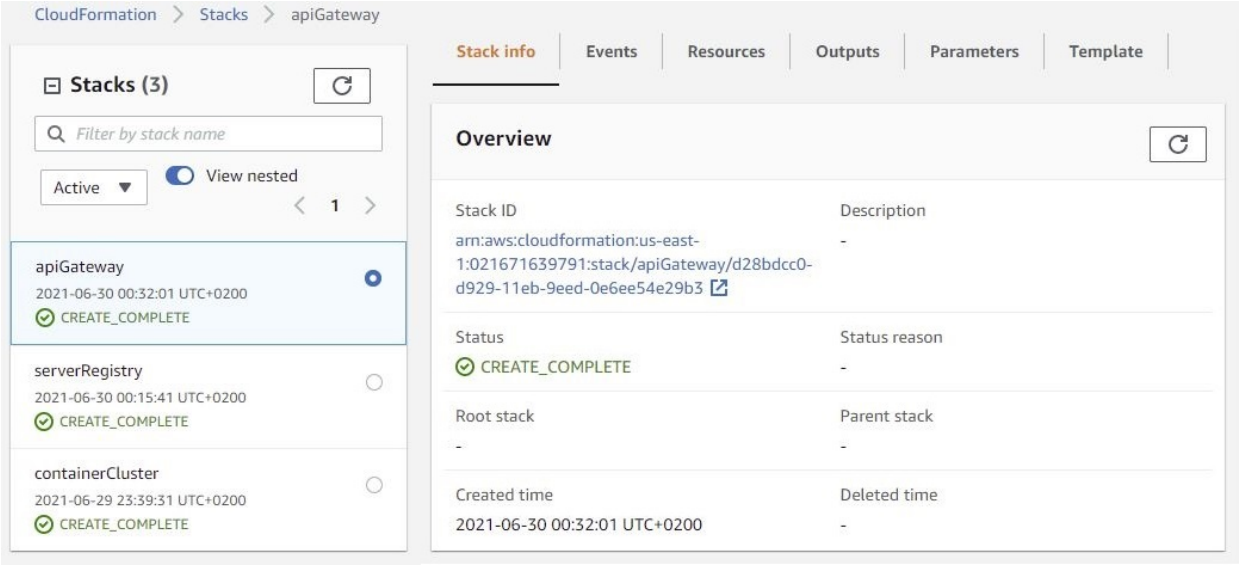


Figura B.11: Cloudformation: Despliegue completo de la receta ApiGateway

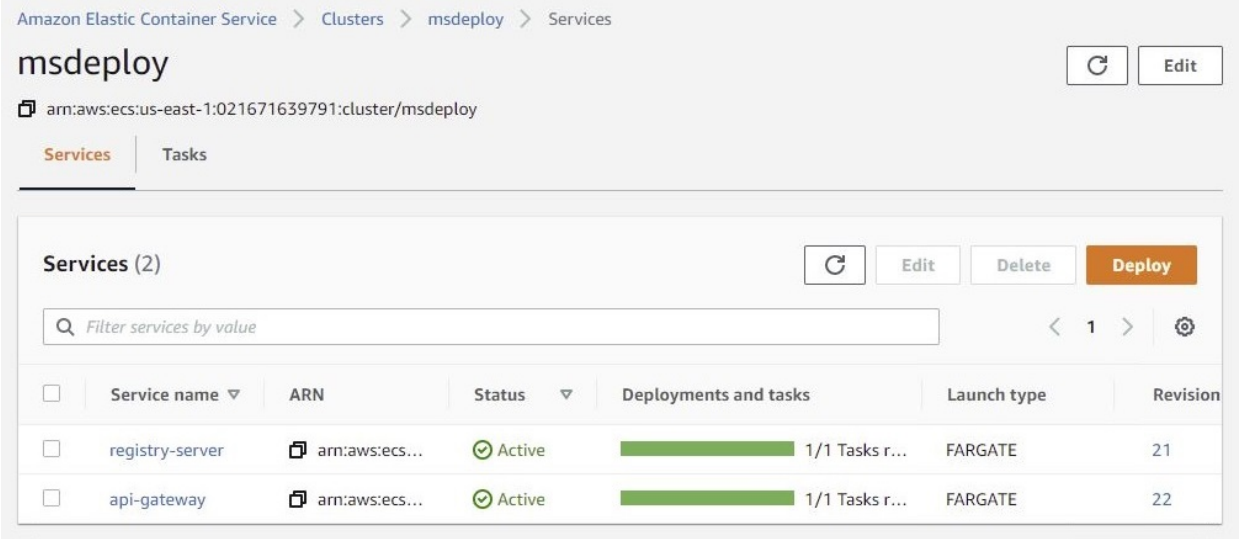


Figura B.12: ECS: Microservicio ApiGateway desplegado en el *cluster*

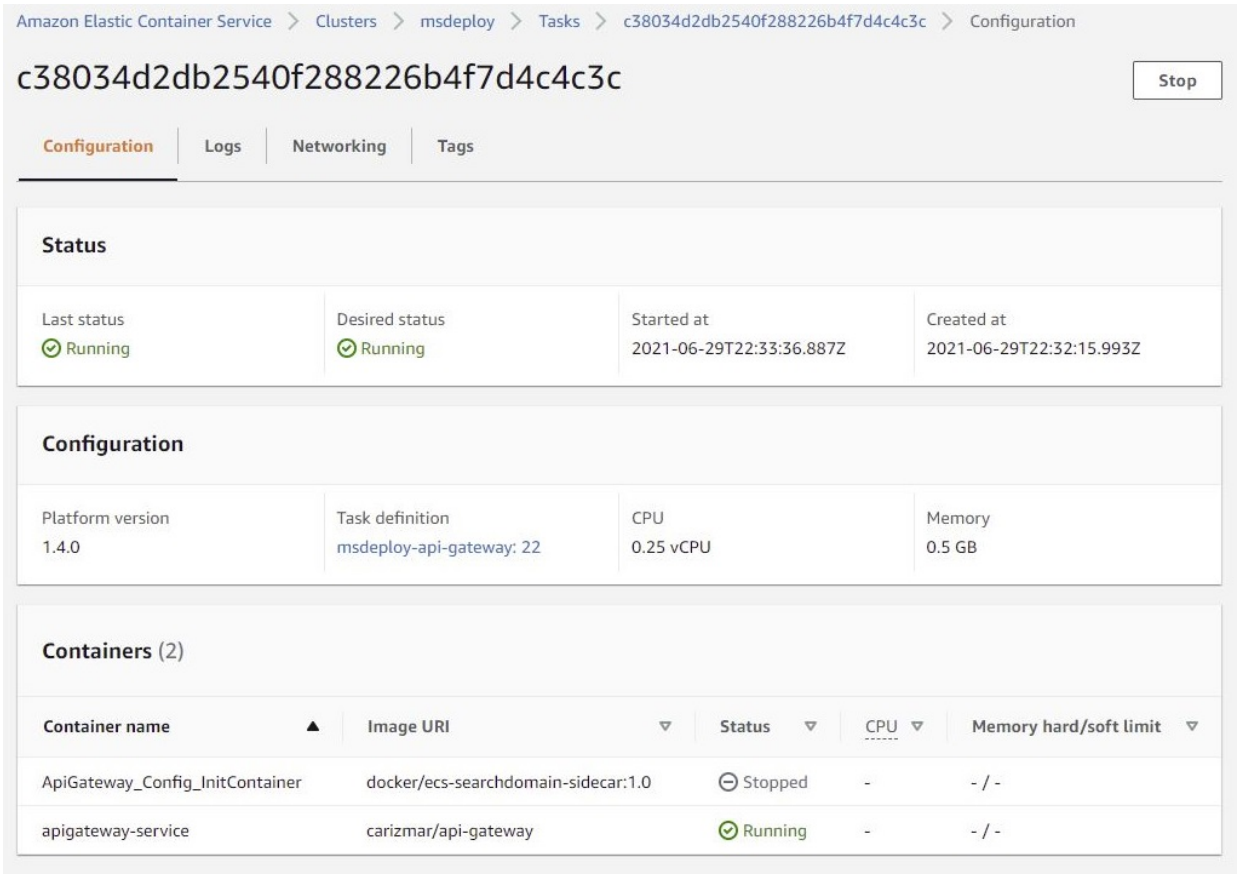


Figura B.13: Fargate: Ejecución de los contenedores *sidecar* y api-gateway en la tarea (nodo)

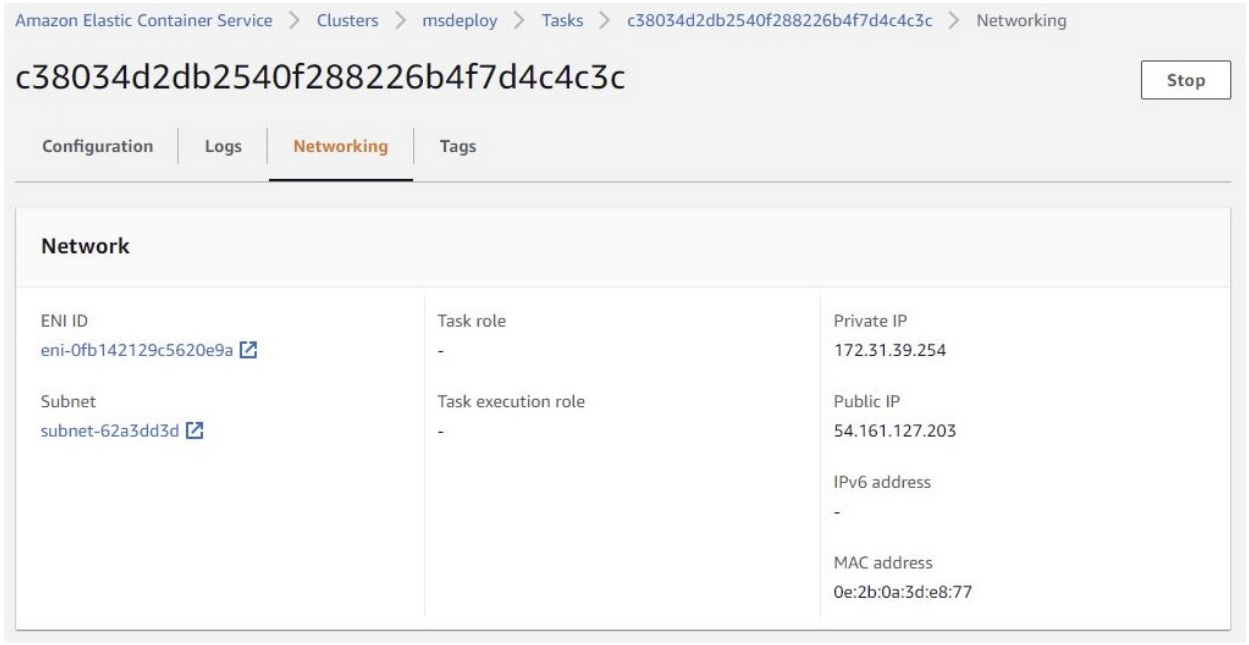


Figura B.14: Fargate: Configuración de red de la tarea (nodo) del ApiGateway

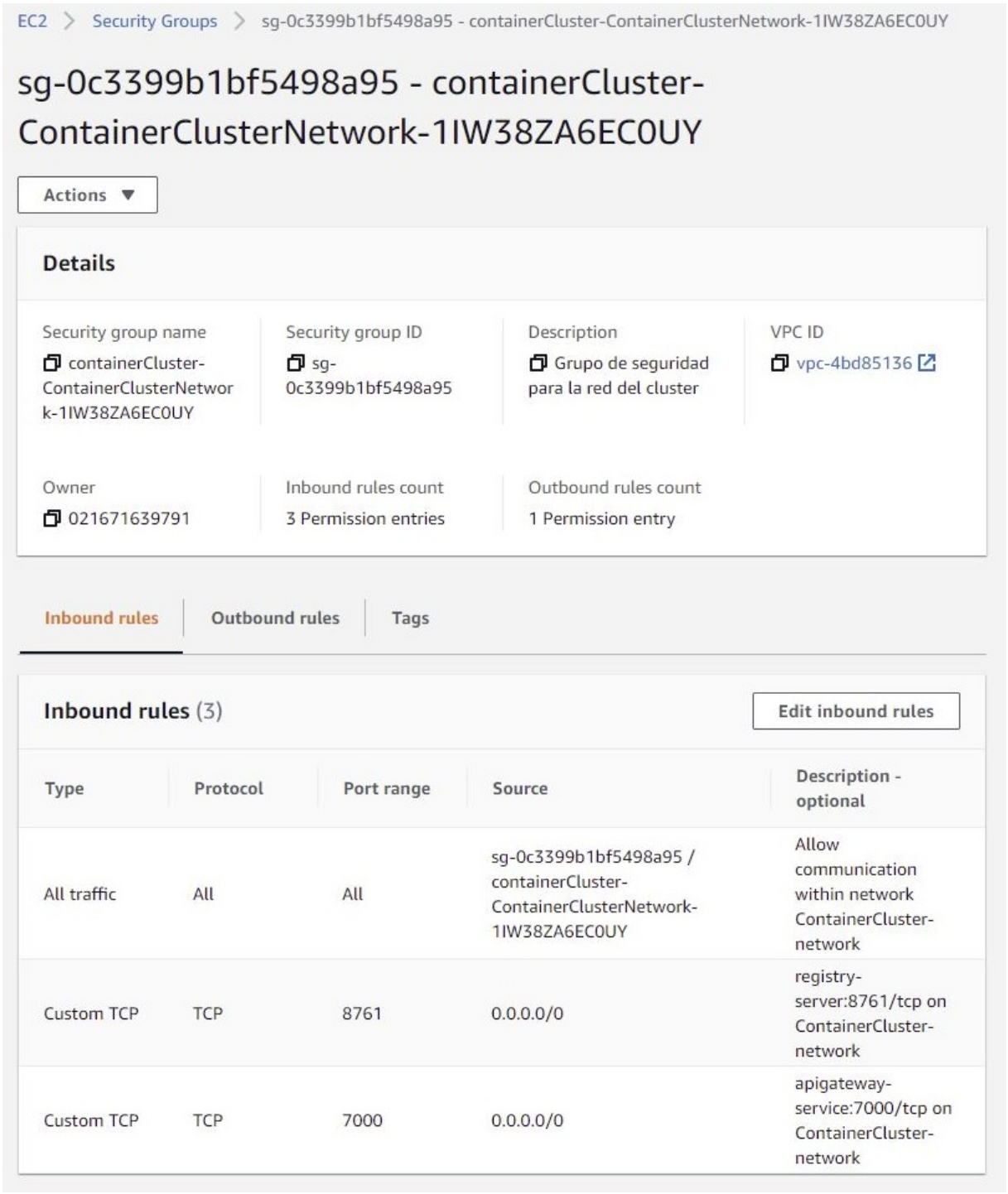


Figura B.15: SG: Configuración de ServiceRegistry y ApiGateway en el grupo de seguridad del *cluster*

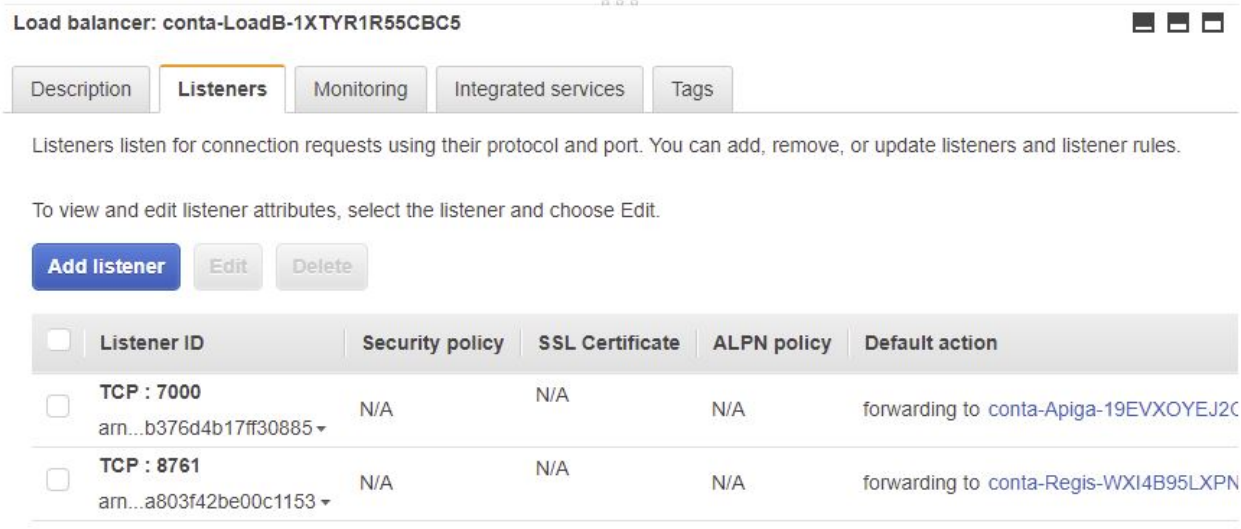


Figura B.16: ELB: Configuración de los Listeners redirigiendo solicitudes a los Target Group

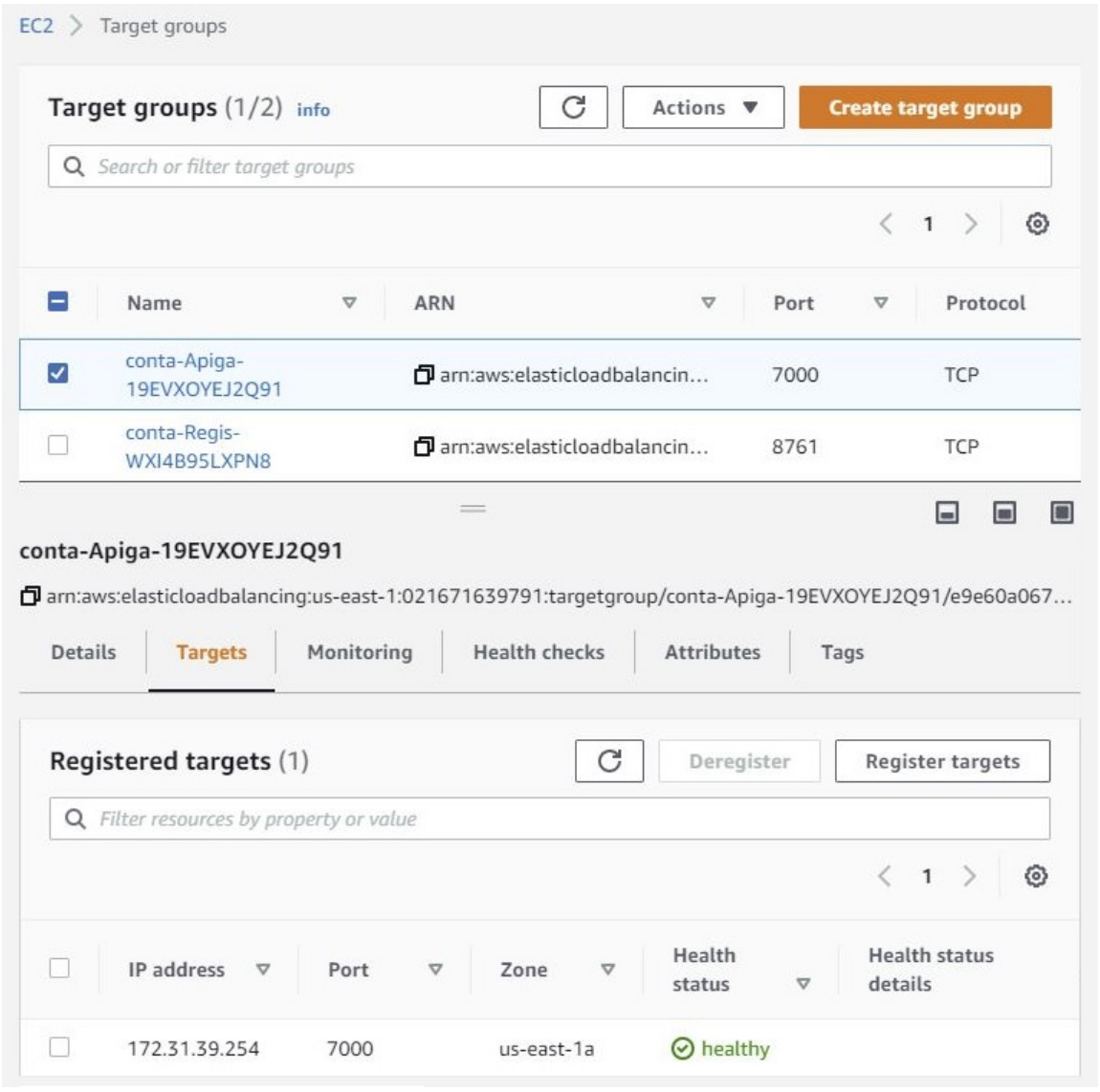


Figura B.17: ELB: Target Groups con las tareas (nodos) Fargate registrados

- **POST. Añadir el microservicio multiplier al espacio administrado**

```
efector.touchpoint:8182/adaptiveSolution
```

Cuerpo de la petición:

```
{
  "id": "multiplier",
  "status": "undeploy",
  "port": 0,
  "ip": "private",
  "image": "carizmar/double-microservice"
}
```

- **POST. Añadir el microservicio trafficLight al espacio administrado**

```
efector.touchpoint:8182/adaptiveSolution
```

Cuerpo de la petición:

```
{
  "id": "trafficLight",
  "status": "undeploy",
  "port": 0,
  "ip": "private",
  "image": "carizmar/traffic-light"
}
```

- **GET. Conocer el estado actual del sistema administrado**

```
sensor.touchpoint:8182/adaptiveSolution
```

Respuesta:

```
{
    "undeployed": [
        {
            "multiplier": 0
        },
        {
            "traffic-light": 0
        }
    ],
    "deployed": [
        {
            "serverRegistry": 1
        },
        {
            "containerCluster": 1
        },
        {
            "apiGateway": 1
        }
    ],
    "inconsistent": [],
    "retired": [],
    "id": "touchpoint"
}
```

- **PUT. Desplegar el microservicio multiplier en el *cluster***

```
efector.touchpoint:8182/adaptiveSolution/service/multiplier
```

Cuerpo de la petición:

```
{
    "status":"deploy"
}
```

- **PUT. Desplegar el microservicio traffic-light en el *cluster***

```
efector.touchpoint:8182/adaptiveSolution/service/traffic-light
```

Cuerpo de la petición:

```
{
    "status":"deploy"
}
```

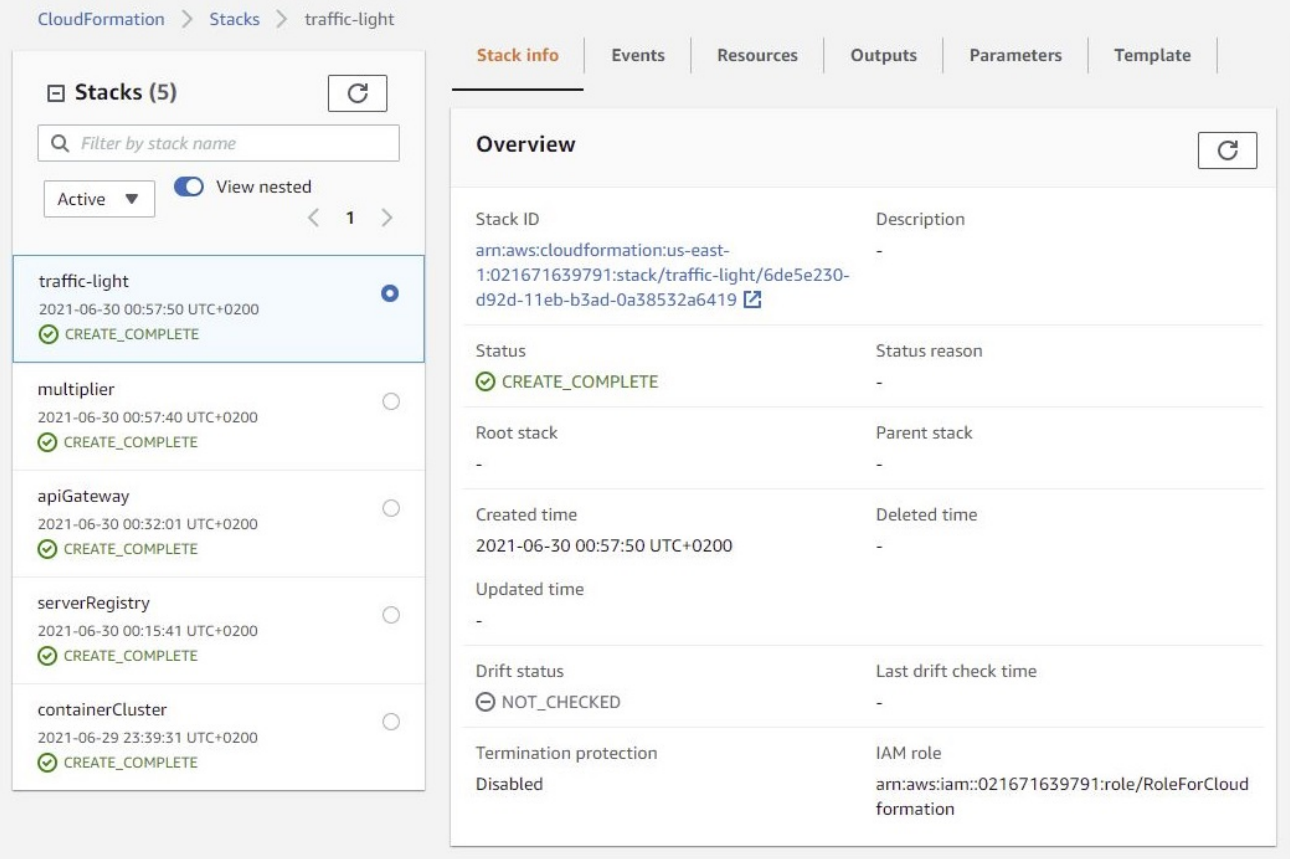


Figura B.18: Cloudformation: Despliegue completa de las recetas multiplier y traffic-light

- **GET. Conocer el estado actual del sistema administrado**

```
sensor.touchpoint:8182/adaptiveSolution
```

Respuesta:

```
{
    "undeployed": [],
    "deployed": [
        {
            "serverRegistry": 8761
        },
        {
            "containerCluster": 0
        },
        {
            "multiplier": 0
        },
        {
            "apiGateway": 7000
        },
        {
            "traffic-light": 0
        }
    ],
    "inconsistent": [],
    "retired": [],
    "id": "touchpoint"
}
```

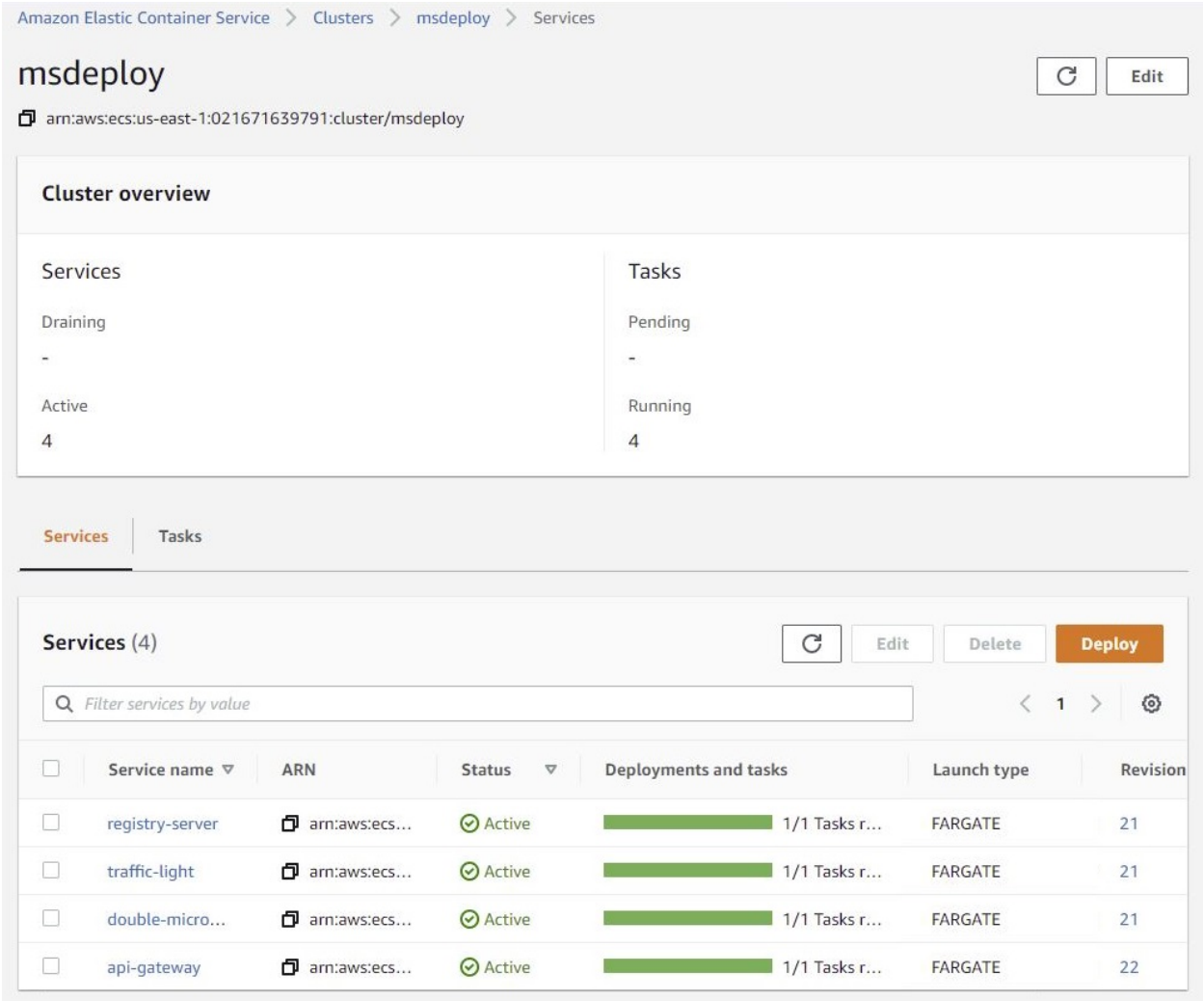



Figura B.19: ECS: Microservicios multiplier y traffic-light desplegados en el *cluster*

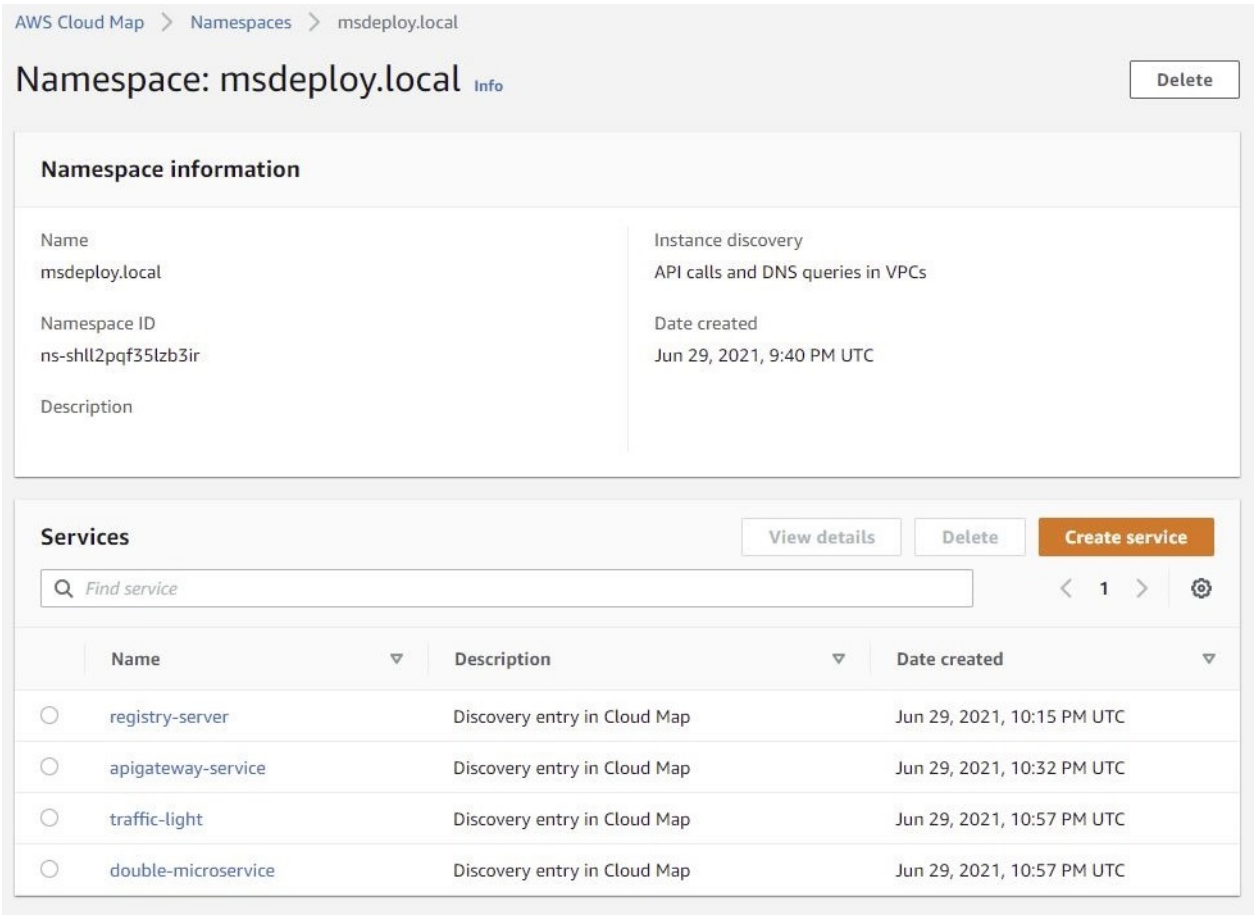


Figura B.20: CloudMap: Espacio de nombres privado actualizado con las tareas Fargate

- **GET. Acceder al serviceRegistry desde el ELB**

```
1 https://conta-LoadB-1XTYR1R55CBC5-ad4eb1e762777131.elb.us-east-1.
      amazonaws.com:8761/
```

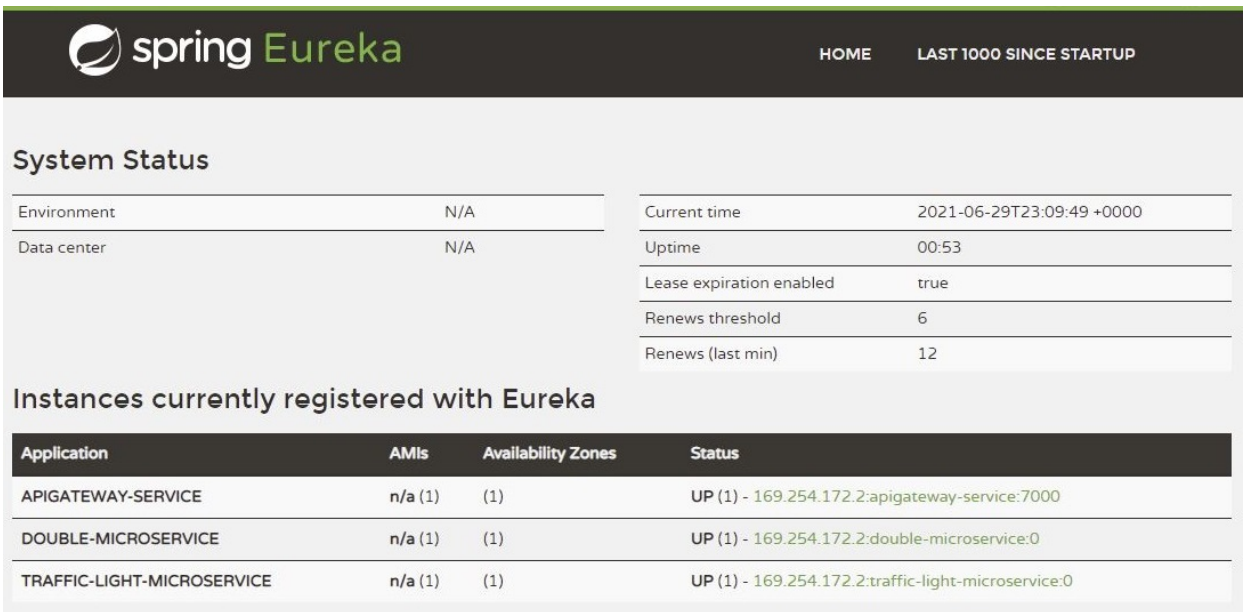


Figura B.21: ServiceRegistry: Acceso al servidor de registros Eureka desde el ELB:8761

- **GET. Acceder al serviceRegistry desde el ELB**

```
https://conta-LoadB-1XTYR1R55CBC5-ad4eb1e762777131.elb.us-east-1.
    amazonaws.com:7000/traffic-light
```

Respuesta:

```
{
    "id":"f1",
    "estado":"Encendido"
},
{
    "id":"f2",
    "estado":"Parpadeando"
},
{
    "id":"f3",
    "estado":"Apagado"
}
```

- **PUT. Retirar el microservicio multiplier del *cluster***

```
    efector.touchpoint:8182/adaptiveSolution/service/multiplier
```

Cuerpo de la petición:

```
{
    "status":"undeploy"
}
```

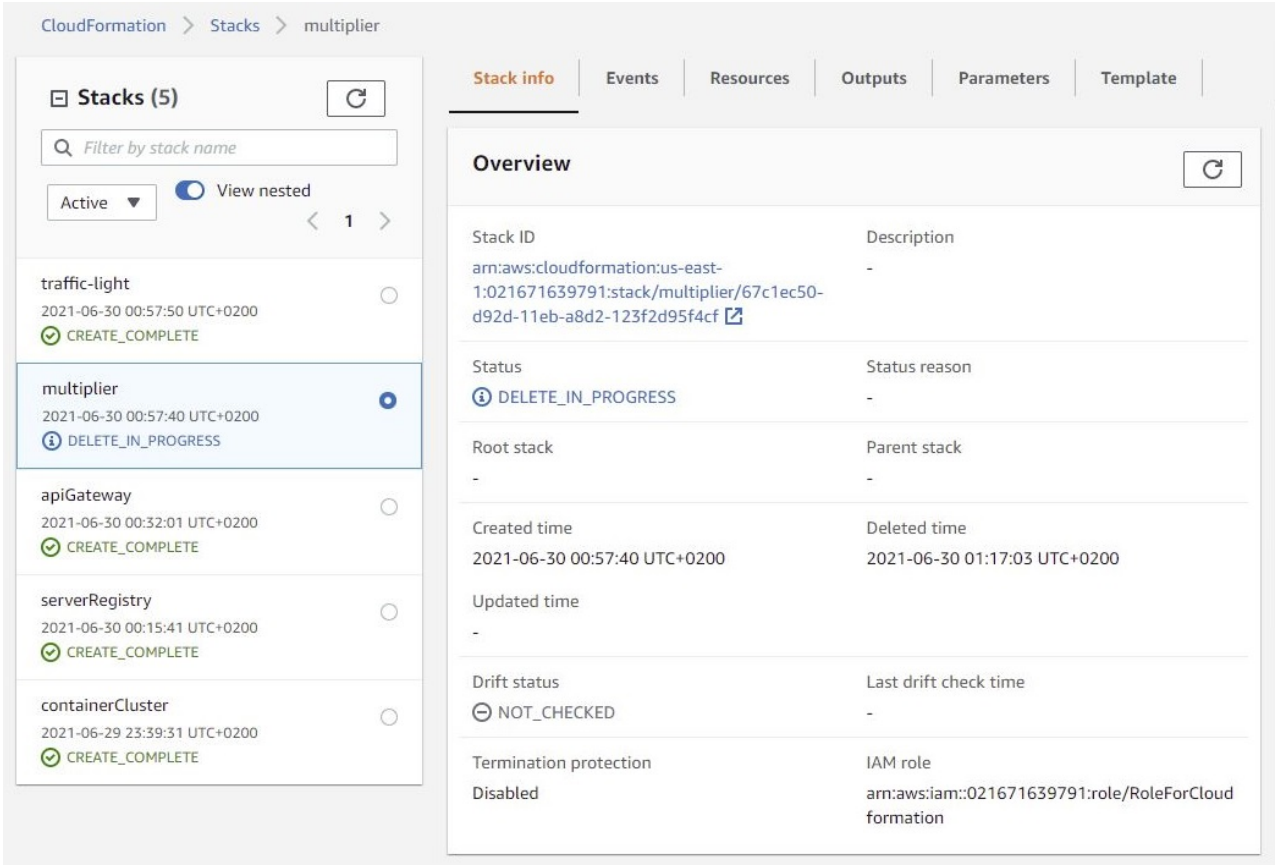


Figura B.22: Cloudformation: Retirar la receta desplegada de multiplier