POSGRADOS

Maestría en

Software con mención en Diseño de Arquitectura de Sistemas

RPC-SO-34-NO.778-2021

Opción de Titulación:

Proyecto de titulación con componentes de investigación aplicada y/o de desarrollo

Tema:

Diseño e implementación de una plataforma para la autogestión de integraciones basadas en el consumo de datos procesados de telemetría vehicular, aplicado a clientes empresariales de la empresa Location World

Autor

Pablo Sebastián Calderón Maldonado

Director:

Christian Merchán Millán

CUENCA – Ecuador

2023

Autor:

Pablo Sebastián Calderón Maldonado
Ingeniero de Sistemas.
Candidato a Magíster en Software, Mención en Diseño de Arquitectura de Sistemas por la Universidad Politécnica Salesiana – Sede Cuenca.
pcalderonm@est.ups.edu.ec

Dirigido por:

Christian Merchán Millán
Ingeniero en Computación.
Magíster en Sistemas de Información.
cmerchan@ups.edu.ec

Todos los derechos reservados.

Queda prohibida, salvo excepción prevista en la Ley, cualquier forma de reproducción, distribución, comunicación pública y transformación de esta obra para fines comerciales, sin contar con autorización de los titulares de propiedad intelectual. La infracción de los derechos mencionados puede ser constitutiva de delito contra la propiedad intelectual. Se permite la libre difusión de este texto con fines académicos investigativos por cualquier medio, con la debida notificación a los autores.

DERECHOS RESERVADOS
2023 © Universidad Politécnica Salesiana.
CUENCA – ECUADOR – SUDAMÉRICA
Pablo Sebastián Calderón Maldonado
Diseño e implementación de una plataforma para la autogestión de integraciones basadas en el consumo de datos procesados de telemetría vehicular, aplicado a clientes empresariales de la empresa Location World

Índice general

Índice de Figuras V

Índice de Tablas VII

Resumen VIII

Abstract VIII

1. Introducción 1
- 1.1. Descripción general del problema 2
- 1.2. Alcance 4
- 1.3. Justificación 6
- 1.4. Objetivos 6
  - 1.4.1. Objetivo general 6
  - 1.4.2. Objetivos específicos 6
- 1.5. Organización del manuscrito 7

2. Marco Teórico 8
- 2.1. Plataformas de Integración como Servicio 9
- 2.2. Soluciones *iPaaS* bajo licencia y open-source 10
  - 2.2.1. Spring Integration 11
  - 2.2.2. Apache Camel 11
  - 2.2.3. Mulesoft 12
- 2.3. Arquitectura basada en microservicios para las *iPaaS* 14
  - 2.3.1. Tecnologías para el manejo de microservicios 17
- 2.4. Criterios de Seguridad y/o Privacidad de la información . . . 19
- 2.5. Infraestructura como código 21
  - 2.5.1. Terraform 21
  - 2.5.2. Pulumi 23

3. Desarrollo de la Solución 26
- 3.1. Diseño de arquitectura lógica 27
  - 3.1.1. API Admin & API Integration Management 28
  - 3.1.2. Integration Generator 29

3.1.3. Integration Scheduler . . . 32
3.2. Repositorio y esquema de datos . . . 33
3.3. Diseño de Arquitectura tecnológica . . . 35
3.3.1. Arquitectura tecnológica en infraestructura local . . . 35
3.3.2. Arquitectura tecnológica en la nube . . . 38
3.4. Personalización de *Apache Camel K* . . . 40
3.5. Infraestructura del *iPaaS* en la nube . . . 40
3.5.1. Estructura de Directorios . . . 44
3.5.2. Seguridad de la infraestructura . . . 44
3.6. Síntesis del capítulo . . . 46

4. Resultados y discusión 48
4.1. Contexto de la prueba de concepto . . . 49
4.2. Pruebas funcionales de la plataforma . . . 49
4.2.1. Prueba funcional del despliegue de plataforma . . . 50
4.2.2. Prueba funcional de autogestión de integración . . . 53
4.2.3. Pruebas funcional de funcionamiento del *iPaaS* . . . 57
4.3. Análisis resultados y discusión . . . 58
4.3.1. Control de errores del *iPaaS* . . . 60
4.4. Síntesis del capítulo . . . 61

5. Conclusiones 62
5.1. Conclusiones . . . 63
5.2. Recomendaciones . . . 64

Glosario 68

Índice de Figuras

2.1. Diagrama alto nivel de *Apache Camel K* (Apache-Software-Foundation) 13
2.2. Diagrama de Capas de la herramienta Any Platform (luisclausin@gmail.com, 2020) 14
2.3. Estructura de una máquina virtual, un contenedor y un orquestador de contenedores (Campbell) 19
2.4. Implementación de *IaC* con VCS, pruebas y despliegue en nube 22
2.5. Flujo de ejecución de alto nivel de *Terraform* 23
2.6. Retroalimentación enriquecida en *IDE*, en la definición de *IaC* con *Pulumi* 24

3.1. Arquitectura lógica de in *iPaaS* autogestionable 27
3.2. Diagrama de bloques de servicios *APIs* 29
3.3. Diagrama de secuencia de servicio *API* Integration Management 29
3.4. Diagrama de secuencia de servicio *API* Admin Management . 30
3.5. Diagrama de bloques del servicio Integration Generator . . . 30
3.6. Diagrama de secuencia de servicio Integration Generator . . . 31
3.7. Diagrama de bloques del servicio Integration Scheduler 32
3.8. Diagrama de secuencia de servicio Integration Generator . . . 33
3.9. Diagrama Entidad Relación del modelo de datos 34
3.10. Diagrama de clases del dominio 36
3.11. Arquitectura tecnológica en infraestructura local 37
3.12. Arquitectura tecnológica con infraestructura en la nube . . . 39
3.13. Componente personalizado de *Apache Camel K* 41
3.14. Esquema de Infraestructura como Código dispuesta para el *iPaaS* 42
3.15. Esquema de *namespaces* propuesto en *Kubernetes* 45
3.16. Ejemplo de esquema terraform que define elementos de seguridad en *Kubernetes* 46

4.1. Componentes y recursos *iPaaS* corriendo en *Google Kubernetes Engine* 52

4.2. Entidades *Datasource* e *Integration* creadas en repositorio principal . . . 54
4.3. Definiciones de integración creadas y cargadas a almacenamiento principal . . . 55
4.4. Procesamiento de archivos de integración y carga en *Kubernetes* 56
4.5. Envío de mensaje a través del sistema de colas . . . 57
4.6. Procesamiento y recepción de mensaje por *iPaaS* y destino . 58
4.7. Integración continua de las integraciones con *Google Cloud Build* y *Github* . . . 59
4.8. Control de errores mediante servicios de la nube . . . 60

Índice de Tablas

2.1. Escenarios de integración 11
2.2. Aplicabilidad de *IP* basada en microservicios 17
2.3. TOP 5 en costos por brechas de seguridad 2022-2021 20

3.1. Recursos *open-source* de plataforma *iPaaS* 38
3.2. Agrupamiento y estructura de directorios para recursos Terraform de la plataforma *iPaaS* 43

Resumen

A día de hoy, el volumen de datos generados por distintos servicios y nuevas fuentes de datos, como dispositivos de tipo *IoT* por ejemplo, y la estructura heterogénea de los mismos, ha crecido de manera exponencial. Esto ha conllevado a muchas empresas en invertir gran cantidad de su tiempo, esfuerzo humano y dinero, al desarrollo a medida y posterior mantenimiento, de procesos que permitan integrar las distintas fuentes de datos, con muchos de los sistemas que al final del día, generan y entregarán la información relevante a los usuarios dueños de los datos.

Con el surgimiento de nuevos enfoques de arquitectura en el diseño de sistemas y basadas en microservicios, la aparición de nuevas soluciones tecnológicas en la generación y orquestación de servicios contenerizados, como *Docker* o *Kubernetes*, la gestión automatizada de infraestructuras a través de herramientas de infraestructura como código, y el grado de madurez adquirido por algunos marcos de trabajo en la gestión de plataformas de integración empresariales, ahora es posible plantear soluciones de sistemas que se ajusten a los nuevos requisitos impuestos por la acelerada transformación con la que los datos van experimentando cada día.

Es así que, la propuesta del desarrollo del siguiente trabajo se enfoca en el diseño de una alternativa de solución para la implementación de una plataforma de integración como servicio y enfocado en la autogestión, utilizando para ello la definición de arquitecturas basadas en microservicios, y apoyado en un conjunto de marcos de trabajo, herramientas y tecnologías descritas previamente; alcanzando así, la optimización en uso de recursos humano, tiempo y costos en las empresas.

Abstract

Currently, there is an exponential increase in the volume of data and their heterogeneous representations, produced by several new services and data sources, such as *IoT* devices. Many companies have incurred significant investments in human resources, time, and money, in order to develop and maintain new services for integrating different data sources.

With the rise of microservices-based architectural approaches for designing systems, the availability of new technology solutions to orchestrate container-based services like *Docker* and *Kubernetes*, the ability to automate the computational infrastructure with infrastructure-as-a-code tools, and how matured some enterprise integration frameworks are over time, all of them have contributed in searching new proposal of system designs that follow the accelerate transformation perceived by the data every day.

Therefore, the following project aims to design a better solution that implements an integration platform as a service with self-management capabilities, with the benefit of reducing costs involved in human resources and time.

Capítulo 1

Introducción

La evolución de la información basada en telemetría ha tenido un crecimiento importante, tanto en los diferentes tipos de información, como en el volumen de datos generados. Desde la captura de datos de ubicación satelital *(Global Positional System GPS)*, durante los años 60 y por producto de la Guerra Fría, hasta la actualidad, en donde la diversidad de fuentes de los datos es evidente, desde teléfonos, computadores, asistentes virtuales como Alexa de Amazon o Siri de Apple, dispositivos médicos, accesorios del hogar, dispositivos de seguimiento y control físico, entre otros. Según (Balasubramanian et al., 2021), se estima que para 2025, se habrá alcanzado un trillón de dispositivos conectados y transmitiendo datos a cada momento. Este gran abanico de tipos de información ha catapultado la transformación y el surgimiento de nuevos modelos de negocio en la industria. Las empresas empiezan a aprovechar la riqueza inhereten de esta información para la constitución de nuevos productos y servicios con un alto valor para un mercado cada vez más hambriento y exigente a la hora de consumirlos. Uno de estos mercados es el de servicios basados en la telemetría vehicular, y representado por empresas que ofrecen diversos servicios como la gestión de flotas, el manejo de seguros a medida, servicios de análisis para reducción de la siniestralidad, reducción de la huella de carbono, etc. De este mercado, se prevee una proyección de 400 millones de vehículos conectados en carretera, para el 2025, y cerca del 96 % de todo el parque automotor a nivel mundial estará conectado, según el análisis establecido en (Abravanel, 2021).

1.1. Descripción general del problema

Partiendo del inminente crecimiento en volumen, como en diversidad de la información en los distintos mercados mundiales, y más aún, dentro del segmento de la movilidad, es evidente las nuevas oportunidades de modelos de negocio que pueden surgir para satisfacer el apetito progresivo de los consumidores por nuevos productos y servicios que aporten valor significativo a su calidad de vida.

Desde el año 2008, empresas del medio visionaron el potencial que existía en la información geo referenciada sobre diversas entidades de interés, y que éstas se encuentren dispuestas sobre un área geográfica de importancia. En ese momento, se contaba ya con los primeros servicios de búsqueda y ubicación, basado en sitios de interés sobre un mapa en particular, donde el usuario podría interactuar con la misma a través del canal digital del momento como lo es la web. A pesar de ser información que no cambiaba con el tiempo, aportó de mucho valor a los usuarios a la hora de optimizar su tiempo encontrando cosas. Más adelante, se evidenció la oportunidad de explotar el valor que daba la información geo referenciada y llevarlo a proporcionar servicios basados en ubicación al instante; permitiendo de esta manera el surgimiento de nuevos productos y servicios basados en dicha mecánica.

Entre las empresas visionarias que experimentaron estos cambios graduales en la manera de hacer negocios con información geo referenciada, está Location World. Una empresa enfocada en apalancar su modelo de negocio a partir de los datos geo referenciados, más la captura de datos generados por dispositivos de tipo Internet de las cosas *(Internet of Things IoT)*, que se instalaban en los vehículos. Sus servicios han abarcado desde la búsqueda y ubicación de puntos de interés, hasta la ubicación de personas y vehículos en movimiento, llegando a disponibilizar de funcionalidades al segmento de flotas en sus operaciones de transporte nacional. Gracias a la evolución que han tenido sus servicios a razón del volumen y la diversidad de información recopilada, su portafolio de clientes ha crecido en proporción, llegando a copar mercados en la región como Colombia, Perú, Argentina, Chile, México y España; así como también, la diversidad de clientes con las que se ha apalancado, desde consumidores finales de un producto o servicio, hasta empresas y socios estratégicos que han aprovechado los servicios de Location World para potenciar sus propios modelos de negocio. En este punto, Location World empezó a levantar diversos proyectos de integración con sus clientes empresariales, para que a través de ellos, se puedan ofrecer nuevos modelos de servicios hacia mercados altamente potenciales para su adquisición y consumo. Por ejemplo, empresas de la línea de seguros, podían ajustar sus modelos de negocios de venta de seguros de vehículos a medida, en función de la información basada en el comportamiento de conducción del consumidor, y que era proporcionada por Location World a través de uno o

varios servicios de integración.

Ahora bien, con el objetivo de seguir manteniendo un nivel competitivo importante, dentro un segmento de mercado en donde nuevos actores se van involucrando en el juego de ofrecer más servicios a menor costo, y al mismo tiempo, comandar el crecimiento de las operaciones que garanticen un alto grado de calidad en los servicios proporcionados, se requiere una importante inversión financiera, tecnológica y humana. Sin embargo, con la llegada de la pandemia en el 2020, la realización de tal inversión se suspendió, y el enfoque de la empresa cambió drásticamente para mantener sus servicios, a un grupo limitado pero clave de clientes, y al mismo tiempo, sacrificando parte del recurso laboral y tecnológico que permita controlar el flujo financiero de la empresa, ante la crisis mundial.

Durante la crisis, la empresa tuvo que reinventarse a tal punto para seguir manteniendo a sus clientes claves, y al mismo tiempo, abarcar a nuevos clientes de gran envergadura en otros países, no obstante, manteniendo un conjunto humano y tecnológico reducido para responder ante el nuevo desafío. A partir de ese momento, la mayor cantidad de proyectos que se desarrollaban en la empresa (alrededor del 70 %) correspondían a nuevas integraciones que los clientes empresariales existentes requerían para reinventar sus modelos de negocio y sobrevivir a la crisis. El hecho fue, que el nivel de atención de los nuevos requerimientos sobre integraciones ya no era el adecuado, debido a la limitación en cuanto al número de desarrolladores para ejecutarlos, y en consecuencia, provocaba serios retrasos en las entregas de los clientes que, al final del día, provocaban drásticos cambios en los requerimientos para ajustarse a los tiempos establecidos, con el subsecuente re trabajo, o peor aún, terminaban desechando el producto terminado. Lo que una integración tomaba en promedio alrededor de 4 días en su desarrollo hasta su puesta en producción, ahora se tomaba cerca de 15 días para su culminación, represando una cantidad de otros requerimientos de similares características. Por otro lado, la gran cantidad de servicios de integración creció a tal punto, que características de los sistemas, como la mantenibilidad, la observabilidad, la escalabilidad, se convirtieron en un desafío, a la hora de proveer el soporte requerido en las mismas.

A este problema, se suma el hecho de que el surgimiento de nuevos proyectos, enfocados a crear disrupción en el mercado, con la materialización de nuevos productos y servicios, no podían ser desarrollados en su totalidad, debido a la falta de mano de obra que los atienda y que no se encuentren enfocados en el desarrollo de integraciones. Al final del día, la empresa terminaba alterando la prioridad de estos proyectos, colocándolos en espera o en otros casos, los desechaban completamente. En consecuencia, debido a la estrategia planteada por la empresa para suplir las necesidades cada vez más comunes de parte de sus clientes empresariales con la necesidad de información, se terminó experimentando una pérdida de oportunidades de negocio que hubiesen permitido mantener su competitividad en el mercado, y

que al final del día, fueron aprovechados por la propia competencia regional.

1.2. Alcance

Considerando la problemática por la que atraviesa la empresa Location World, al igual que otras empresas del medio, principalmente como consecuencia de la crisis mundial por el surgimiento reciente de la pandemia, es evidente la necesidad de incorporar una alternativa de solución acorde a esta nueva realidad. Si bien se podría determinar que la solución más efectiva en este caso se encaminaría en la adquisición de mayor y mejor talento humano para suplir la desproporción a la hora de gestionar los ineludibles desarrollos, el ajuste financiero sin embargo, producto de la pandemia, imposibilita la ejecución de dicha acción. Otra propuesta de solución podría encaminarse en el levantamiento de un nuevo canal digital de servicio que permita equilibrar la atención requerida para las empresas demandantes de la información, una práctica muy utilizada en las empresas en la búsqueda de efectividad a la hora de cumplir con necesidades emergente de sus clientes, a través del uso de recursos tecnológicos y con una optimizada capacidad de talento humano.

Si se profundiza en la segunda propuesta de solución a la problemática descrita, el nuevo canal digital de servicio estaría habilitado a partir de la construcción de un sistema, con una infraestructura subyacente optimizada a los costos establecidos dentro de un presupuesto previamente definido. Otro aspecto a considerar para este nuevo canal de servicio radica en el factor tiempo, indispensable tanto para el despliegue del servicio como para el inmediato uso de parte del cliente final. A fin de cumplir con esta expectativa, la incorporación de procesos de automatización al ciclo de vida del nuevo canal de servicio, permitirán reducir sustancialmente los tiempos de espera y de uso del mismo hacia sus consumidores.

Ahora, para que este canal de servicio sea apto para su consumo, se puede contemplar dos escenarios para su disponibilización. El primer escenario contempla el levantamiento del nuevo canal digital por parte de Location World, a través de la ejecución de una serie de pasos mínimos y con el uso de la infraestructura adecuada para tal efecto, descartando de esta manera, cualquier desarrollo a medida que demande un esfuerzo de trabajo y tiempos considerables. De este escenario, sin embargo, surge un problema, debido a la dependencia operativa de parte de Location World para cumplir con la necesidad del cliente, y que en muchos casos, puede no ajustarse a los tiempos esperados del cliente para su uso.

El otro escenario que se contempla, se presenta en entorno a la autogestión del canal de servicio, en donde el actor demandante de la información, se convierte en la entidad clave al momento de levantar el canal de atención requerido para su consumo, con el apoyo de la infraestructura y

servicios necesarios provistos por Location World para tal efecto.

En consecuencia, la necesidad emergente de Location World de ajustar sus operaciones al ritmo demandado por sus clientes y de acuerdo con la situación actual que experimenta la empresa, puede ser resuelta con el apoyo de recursos tecnológicos y nuevas tendencias en el diseño y desarrollo de servicios que constituyan el nuevo canal digital y optimizado de atención, junto con la posibilidad de incluir el proceso de autogestión necesario de parte del cliente final, para su disponibilización en el tiempo esperado y sin dependencia alguna más que la infraestructura subyacente.

Es así que, la propuesta de este proyecto de titulación se enmarca en el levantamiento de una plataforma digital, mediante la implementación de una serie de servicios requeridos, diseñados a partir de arquitecturas de sistemas basadas en microservicios, en donde la mantenibilidad de cada componente de la plataforma está garantizada por el aislamiento, la modularidad y la reusabilidad inherente en el diseño. Así mismo, la escalabilidad de dicha plataforma sería rápida y óptima al mismo tiempo, en parte, por la facilidad con la que parte de la plataforma y su infraestructura subyacente se expandiría o se reduciría, en función de la demanda de utilización a lo largo del tiempo. Para el desarrollo de los servicios que compongan dicha plataforma digital, se aprovechará en mayor o menor grado, el uso de soluciones de código abierto *(open-source)*, cuya sólida y madura estructura, aportaría en la homologación de los componentes a ser desarrollados.

Por otro lado, la infraestructura necesaria sobre la que correrá la plataforma, será desplegado con la ayuda de tecnologías que simplifiquen la orquestación de sus componentes y la subsecuente mantenibilidad; y todo ello, sobre soluciones de cómputo en la nube, como el servicio otorgado por *Google Cloud Platform (GCP)* por ejemplo, proveedor principal de infraestructura de Location World. Esta elección favorecerá por un lado, en el control del presupuesto que se destine para este requerimiento, gracias a la capacidad de pago por uso o por compromiso *(commitment)* que se disponen en la nube; y por otro parte, se trasladaría la complejidad del mantenimiento sustancial de la infraestructura hacia el proveedor exclusivamente.

En cuanto a la propuesta enfocada en la autogestión, se propone el desarrollo y su consecuente disponibilización, de una interfaz de servicio *Application Programming Interface (API)*, con un estilo arquitectónico de tipo *REST*, con las configuraciones esenciales que definan el canal de comunicación a establecer entre la nueva plataforma de servicio de Location World y la infraestructura del cliente final. El uso de servicios *API REST* están justificados por su naturaleza uniforme, de fácil interacción, gran escalabilidad, con gran aporte en enriquecer la experiencia de uso.

Por último, se complementa a la propuesta el uso de tecnologías *open-source* para la automatización de la plataforma e infraestructura subyacente que facilite su mantenibilidad en el corto, mediando y largo plazo, con el talento humano mínimo para ello.

1.3. Justificación

La presentación de la propuesta de solución a la problemática de Location World se sustenta en los siguientes argumentos. Por un lado se encuentra la limitada capacidad de talento humano para mantener las actividades de desarrollo y operación al ritmo en que se reciben los requerimientos de nuevos servicios de integración. La factibilidad de contar con una plataforma que permita la construcción inmediata de dichas integraciones, bajo una misma arquitectura, equilibrará el esfuerzo de trabajo del mismo equipo para el volumen de requerimientos y nuevos servicios de integración desplegados.

Por otro lado, el uso de soluciones de cómputo en la nube para suplir necesidades de infraestructura es ventajoso a la hora de optimizar los costos de inversión requeridos para tal efecto, con la consecuente adquisición del soporte necesario en términos de disponibilidad, mantenibilidad, seguridad, y un nivel de servicio adecuado.

Finalmente, la propuesta de autogestión de las integraciones por parte de los cliente empresariales, ayudará a reducir drásticamente los tiempos y esfuerzos de los desarrolladores empleados para dicha tarea, y se los enfocará estratégicamente hacia el desarrollo y ejecución de nuevos proyectos destinados a impulsar al negocio y recuperar la competitividad en el mercado de servicios de movilidad.

1.4. Objetivos

1.4.1. Objetivo general

Diseñar e implementar una arquitectura de solución para la construcción y despliegue de Plataformas de Integración, mediante el uso de arquitecturas basado en microservicios y el apoyo de herramientas y marcos de trabajo *open-source*, para la automatización en la construcción y despliegue de servicios para la entrega de datos procesados de telemetría vehicular.

1.4.2. Objetivos específicos

- Analizar el estado del arte sobre servicios en la nube de tipo *iPaaS* y de herramientas *open-source* que las sustenta, así como arquitecturas basados en microservicios, mediante el estudio y revisión de documentos científicos y no científicos, para formular las bases sobres las que se postulará la propuesta de arquitectura.

- Evaluar la aplicabilidad de un marco de trabajo *open-source*, mediante el desarrollo de una prueba de concepto enfocada en los requisitos mínimos del proyecto, para ser utilizado en el diseño e implementación de la plataforma de integración.

- Establecer los requerimientos de arquitectura propuestos, mediante la elaboración del diseño del modelo de datos, clases, componentes, etc, necesarios para llevar la implementación de la plataforma de integración.
- Elaborar una prueba de concepto de un *iPaaS*, usando un proveedor de infraestructura en la nube como lo es *GCP*, para confirmar o refutar la validez del diseño de arquitectura propuesto.
- Automatizar el levantamiento de la infraestructura de *iPaaS* requerida, mediante el uso de herramientas de tipo Infraestructura como código *(Infrastructure as a Code IaC)*, tales como *Terraform* o *Pulumi*, para disponibilizar el entorno de ejecución de la *iPaaS*.

1.5. Organización del manuscrito

Para el desarrollo de este proyecto, se ha propuesto la siguiente organización de contenido. En el Capítulo 2, se expone un resumen del conocimiento existente acerca de distintas soluciones enfocadas a proveer servicio de integración en las empresas. En el Capítulo 3, se presenta una propuesta de arquitectura basada en microservicios para el montaje de un *iPaaS*, junto con detalles tecnológicos para su implementación. En el Capítulo 4, se presentan los resultados obtenidos a partir de una prueba de concepto establecida y se definen las conclusiones y recomendaciones de este trabajo de titulación, en el Capítulo 5.

Capítulo 2

Marco Teórico

Se presenta en este capítulo un resumen del conocimiento existente acerca de las diferentes soluciones propuestas para abordar el problema que plantea el proyecto, y de sus bases sobre las que se fundamentan. En la Sección 2.1 se describen conceptos y características claves de las llamadas Plataformas de Integración *(Integration Platforms IP) iPaaS*. En la Sección 2.2, se exponen algunas soluciones existentes, tanto las que requiere de una licencia, como de uso libre *(open-source)*, para la implementación de plataformas *iPaaS* dentro de una organización, y resaltando las características claves que ayudarán a la hora de elegir la opción más adecuada. En la Sección 2.3 se plantean las Arquitecturas basadas en microservicios como puntales para el desarrollo de *IP*, la presentación de trabajos de investigación existentes en éste ámbito y una reseña de las diferentes herramientas tecnológicas que favorecen en la implementación de las *IP*, bajo ese tipo de arquitecturas. En cuanto a la sección 2.4, ésta se enfoca en resaltar los criterios en términos de seguridad de la información que deben ser considerados durante el funcionamiento de las *iPaaS*. Por último, en la sección 2.5, se puntualizan aspectos claves de modelos de automatización de infraestructura, junto con herramientas existentes que contribuyen para tal propósito, fortaleciendo el proceso de levantamiento de las *iPaaS*.

2.1. Plataformas de Integración como Servicio

Para entender las ventajas del uso de soluciones *iPaaS* en la atención de la interacción entre productos y consumidores de información de diversa índole, es necesario entender las bases y conceptos que las definen.

Según (Freire et al., 2019), y bajo la traducción al castellano, se resalta el concepto de las “Plataformas de Integración como herramientas de software especializados que proveen soporte para el diseño, implementación, ejecución y monitoreo de soluciones de integración“. Normalmente, las *IP* están compuestas de un grupo de tareas atómicas llamadas filtros, los cuales procesan una serie de datos encapsulados en tránsito a través de una serie de canales de mensajes o tuberías *(pipes)*.

Con el nacimiento de la computación en la nube, como el nuevo paradigma en el desarrollo, comercialización y uso de software fuera de los límites de las infraestructuras de las empresas, se produce un efecto de migración de las *IP* hacia máquinas virtuales en la nube, dando origen al nuevo modelo de servicio denominado *iPaaS*. De acuerdo a (Serrano et al., 2014), y bajo la traducción al castellano, se define a las *iPaaS* como “el conjunto de servicios en la nube que permiten a los usuarios el poder crear, administrar, y gobernar los flujos de integración, al conectar una rango amplio de aplicaciones y fuentes de datos sin requerir de instalaciones o mantenimientos de cualquier hardware o middleware“.

Las *iPaaS* están compuestas por componentes primarios como procesos de integración, mapeo de objetos de datos, adaptadores para conectividad entre tipos de aplicaciones, herramientas de soporte a desarrollo como Kits de desarrollo de software *(Software Development Kits SDKs)*. En las *iPaaS*, la presencia de espacios de venta o *(marketplaces)* de plantillas de mapeo de datos, procesos ya definidos de integración, así como de adaptadores pre-construídos y listos de uso, facilitan el despliegue de estas integraciones, bajo un concepto simple del tipo arrastrar y soltar *(drag and drop)* sobre las *iPaaS*.

Para que esto sea posible, es importante destacar que las *iPaaS* juegan un rol importante en economías cuyos modelos de negocio se centran en la entrega de plataforma, permitiendo disponer de una infraestructura para facilitar la transaccionabilidad de operaciones entre las empresas y proveedores de soluciones para las integraciones.

Según se analiza en (Neifer et al., 2021), las *iPaaS* son el intermediario que permite la conectividad entre los proveedores de Software como Servicio *(Software as a Service SaaS)* y los negocios, a través de la disposición de una serie de conectores desarrollados en base a una única interfaz o modelo común; garantizando de esta manera, soluciones de calidad con baja incidencia de error, incrementando seguridad de los datos y proporcionando integridad mediante el soporte de la gobernanza, administración y monitoreo de los datos.

Finalmente, y según (Marian, 2012), los servicios basados en la nube, como los son los *iPaaS*, permiten tanto en tiempo de diseño como de ejecución, la gobernanza de todos los artefactos de integración, los modelos de proceso, composiciones, transformaciones, reglas de enrutamiento, acuerdos de niveles de servicio *(Service Level Agreements SLAs)* y políticas; todo esto con el afán de responder efectivamente a los requerimientos emergentes de los usuarios y bajo una misma infraestructura integrada. Por otro lado, y considerando que las *iPaaS* se despliegan y ejecutan en ambientes virtualizados y ajenos a cualquier infraestructura local sobre la cual se deba invertir a priori, los clientes adquirientes de estas soluciones están sujetos a un modelo de pago a demanda o pago por servicio, para el ciclo de vida del flujo de sus integraciones: desarrollo, pruebas, despliegue y monitoreo.

2.2. Soluciones *iPaaS* bajo licencia y open-source

Con el objetivo de asistir en la implementación de las *iPaaS* y de reducir el esfuerzo y tiempo dedicados para tal fin, existen una serie de productos y servicios presentados bajo distintos modelos de uso, pero que al final del día, convergen en dos únicos tipos: los licenciados y los *open-source*. A diferencia de las soluciones de software licenciadas, las cuales requieren de un pago único o periódico para su uso, un software que es etiquetado como *open-source*, y como se cita en (Freire et al., 2019), bajo la traducción al castellano, "está disponible bajo una licencia que garantiza al usuario los derechos para su uso, cambio y distribución de cualquier tipo[...]".

De la misma manera, se resalta que las soluciones de software de tipo *IP* proveen un lenguaje específico de dominio (*Java*, *C-Sharp*, etc), un kit de herramientas de desarrollo para la implementación de las soluciones de integración, ambiente para pruebas que permitirán la validación e identificación de defectos o errores en las integraciones, tanto a nivel individual como de sistema. Herramientas de monitoreo, enfocadas a la validación en tiempo de ejecución de la operación de las integraciones y la detección de posibles defectos que puedan ocurrir durante el procesamiento de los mensajes. Por último, un sistema de ejecución, con el soporte necesario para la ejecución de las integraciones.

Basados en la investigación de (Palanimalai and Paramasivam, 2015), los aspectos claves a tomar en cuenta al momento de elegir una plataforma de integración en general son las cuestiones de seguridad, escalabilidad, elasticidad, monitoreo y administración. Así mismo, se incorpora el término, bajo traducción textual del inglés, como plataforma múltiples talentos *(multi-talented platform)*, lo que significa que puede soportar diferentes escenarios de integración. Véase Tabla 2.1

Bajo este breve análisis de los aspectos claves y escenarios sobre las que se desenvuelven las *iPaaS*, que ayudarán en el proceso de selección del producto,

Tabla 2.1: Escenarios de integración

Escenarios	**Descripción**
La nube y local	Integración entre soluciones en la nube y aplicaciones existentes en infraestructura local
Nube a nube	Integración entre soluciones y aplicaciones completamente en proveedores de servicios de cómputo en la nube
Entre infraestructuras locales	Integración entre soluciones y aplicaciones en infraestructura local únicamente
Integración eCommerce B2B	Integración entre soluciones de eCommerce empresariales

se puede describir a continuación soluciones de software para el levantamiento de las *iPaaS* mas reconocidas en el mercado.

2.2.1. Spring Integration

Spring Integration no es una solución por si sola de un *iPaaS*. En este caso se trata de una extensión más, dentro de una enorme lista de componentes que complementan al marco de trabajo *open-source* para desarrollo de *Spring*. El núcleo de esta extensión está desarrollada bajo los principios establecidos por los patrones de integración empresarial, mediante el cual, permite el despliegue de componentes primarios para la composición de servicios integradores: mensajes, canales de mensajes, endpoints de mensajes, filtros, ruteadores, divisores, etc. Esta extensión permite incorporar un modelo de comunicación asíncrono y ligero basado en mensajes, entre aplicaciones desarrolladas en *Spring* y sistemas externos, mediante el uso de adaptadores definidos de manera declarativa.

El lenguaje de dominio para la definición de los componentes (sobre todo los adaptadores) es *Java* y el *XML*. Gracias a una dependencia llamada *Endpoints de Integración de Spring*, ésta se puede conectar a otros componentes de otras soluciones de *IP* como *Apache Camel* por ejemplo.

2.2.2. Apache Camel

Según (Onofré, 2015), Apache Camel es un marco de trabajo de mediación *(mediation framework) open-source* y nace como una mejora del proyecto *Apache ServiceMix*, el cual a su vez fue desarrollado en base al marco

de trabajo de *Spring* junto con la especificación de integración de negocio de *Java Business Integration (JBI)*. Las mejoras se centran en soportar más formatos en mensajes que no sean necesariamente *XML* por cuestiones de performance (sobre todo en comunicación en modo stream) tanto en la serialización como deserialización de datos. Adicionalmente, el empaquetado con *JBI* es ineficiente al que *Apache Camel* propone, que no es más que un simple archivo de definición en formato *XML* que contiene toda la definición, a diferencia de *JBI*, en donde se maneja un paquete por endpoint y otro paquete distinto para su definición y configuración.

El núcleo de la solución de software es muy liviano, llegando a pesar cerca de 2 *MB* y contiene todo lo necesario para su ejecución, sin depender de otros frameworks de trabajo. Soporta múltiples contenedores para su despliegue y ejecución como servidores de aplicaciones *J2EE*, *WebSphere*, *WebLogic*, *Apache Tomcat*, incluso con marcos de trabajo como el de *Spring*, como se había descrito previamente. Así mismo, *Apache Camel* maneja distintos lenguajes expresivos para la definición de las reglas y procesos de integración como *Python*, *PHP*, *Ruby*, *Javascript*, *XPath*, entre los más conocidos.

Adicionalmente, se dispone de un ambiente en donde prototipar y probar lógicas de integración, mediante el uso de datos simulados o lo que se conoce como moqueo de datos, sin la necesidad de una dependencia especial.

Dentro del ecosistema de *Apache Camel*, existe un producto denominado *Camel K*, que no es nada mas que el mismo *Apache Camel* pero con capacidad de ser desplegado y ejecutado de manera nativa en entorno *Kubernetes*; una característica bastante notable dentro del soporte para soluciones desarrolladas en arquitecturas basada en microservicios. Dando soporte a este producto, se suman dos más con sus respectivos enfoques. *Apache Quarkus* que no otra cosa que la disposición de los componentes de *Camel* escritos como extensiones de *Quarkus*, una plataforma que ofrece un arranque optimizado y acelerado de los componentes con una baja huella de uso de memoria. *Camel Karavan*, que no es más que un kit de herramientas para el desarrollo de integraciones, a través de una interfaz gráfica para el diseño, configuración de rutas, integración, empaquetado, construcción de la imagen y despliegue en *Kubernetes*.

2.2.3. Mulesoft

El proyecto *Mule* fue concebido en sus inicios como un proyecto de tipo *open-source*, cuya idea fue la de simplificar la vida de los desarrolladores a la hora de diseñar aplicaciones de integración. Su mayor eje conductor fue la necesidad de contar con una solución que fuese modular y ligera, con capacidad de escalar desde un *framework* de mensajería a nivel de aplicación, hasta un bus de servicios altamente distribuible y de rango empresarial. (D'Emic et al., 2014).

Al igual que las otras alternativas descritas previamente, mantiene el

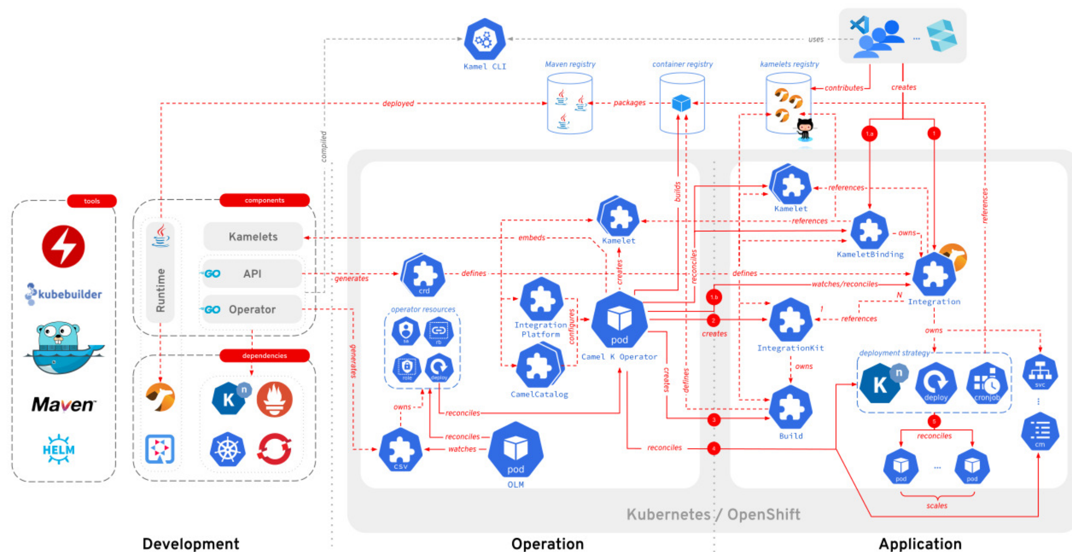


Figura 2.1: Diagrama alto nivel de *Apache Camel K* (Apache-Software-Foundation)

concepto de expansión mediante extensiones acoplables *(pluggables)*. Gracias a estas extensiones, *Mulesoft* es capaz de soportar desde transacciones distribuidas, mecanismos de seguridad nuevos y/o actualizados, al igual que su administración, y la adaptación de tendencias emergentes en el ámbito de cómputo empresarial como repositorios *NoSQL*, grillas de memoria distribuida y hasta protocolos de mensajería ligera como *AMQP* y *ZeroMQ*.

A diferencia de sus competidores *open-source*, el producto evolucionó para convertirse en una solución bajo subscripción exclusivamente, y luego de ser adquirido por parte de la empresa *Salesforce*, cambió su nombre a *Anypoint Platform*. Actualmente, tiene capacidad de despliegue y funcionamiento tanto en la nube pública, como en un centro de datos físico o en ambas locaciones, todo ello en función de las necesidades de seguridad, privacidad de datos y requerimientos legales que una empresa deba cumplir. Esta nueva versión de software incorpora un principio llamado *API-Led connectivity*, que establece una integración basado en servicios de interfaz de programación de aplicación *Application Programming Interface API* reusables que eliminen el contacto directo punto a punto, y que prorporcione un bajo acoplamiento entre las partes para una mejor escalabilidad. Estas APIs se categorizan en 3 grupos i)*APIs* de Experiencia, las cuales son el punto de contacto con los clientes finales, ii)*APIs* de proceso, que interactúan con datos de la capa

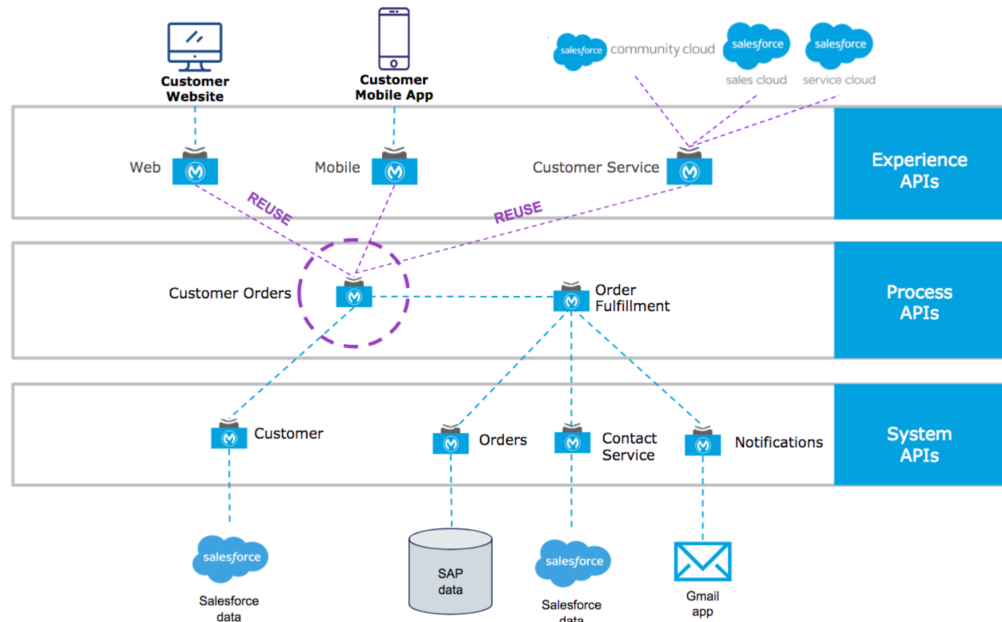


Figura 2.2: Diagrama de Capas de la herramienta Any Platform (luisclausin@gmail.com, 2020)

de sistema y los adaptan para cubrir necesidades de negocio, y iii) *APIs* de sistema, dedicados a procesos de más bajo nivel de conexión con las fuentes de datos. (luisclausin@gmail.com, 2020)

2.3. Arquitectura basada en microservicios para las *iPaaS*

Basándose en (Lewis and Fowler, 2014), la arquitectura basada en microservicios forma parte del paradigma, junto con la arquitectura monolítica, en la ingeniería de software para el desarrollo de aplicaciones empresariales. Una arquitectura de microservicios descompone el dominio del negocio en fracciones más pequeñas, consistentemente delimitadas, con bajo acoplamiento, auto contenidas e independientemente desplegadas, denominados servicios. En la actualidad, muchas de las empresas con renombre mundial han compartido sus experiencias al emprender el proceso de transición de sus plataformas hacia esta propuesta de arquitectura, como lo son *Netflix, Amazon, eBay, Spotify, Airbnb, Uber, Twitter, Coca-Cola,* entre otros.

Ahora bien, en cuanto a la definición de las *IP*, existen estudios previos

que han analizado la incorporación de las arquitecturas de microservicios para el diseño y la construcción de dichas plataformas. Si se toma como referencia el estudio desarrollado por (Nebel, 2019), se evidencia un proceso de investigación y análisis de los atributos claves que poseen las arquitecturas de microservicios y el impacto, tanto positivo como negativo, que experimentarían las *IP* diseñadas bajo estas arquitecturas. El autor, dentro de su análisis, determina que factores como la ubicación de las *IP*, sea en infraestructura local o la nube, así como la cantidad de organizaciones, es decir, si las *IP* son usadas para integrar una única empresa o un conjunto de ellas, la cantidad y madurez de los sistemas a integrar, el volumen y complejidad de los datos, todos ellos pueden determinar el grado de impacto positivo o negativo que las *IP* pueden obtener de estas arquitecturas. Véase Tabla 2.2

Complementario al análisis, se determina que la naturaleza distribuida de las arquitecturas de microservicios, mantienen problemas inherentes como la **complejidad operacional**, la misma que se da por la cantidad de servicios que deben ser gestionados; la **consistencia eventual**, debido a la cantidad de repositorios por número de microservicios que requieren sincronizar la información pertinente; **latencia** relacionada a la comunicación de red entre los servicios, **complejidad** en la supervisión del sistema y la posible identificación de fallas y problemas, son algunas de las cuestiones que requieren de una revisión. A todo problema, sin embargo, existe su respectiva solución o plan de acción mitigadora, y dentro de este contexto, la literatura es variada y completa. Se destaca por ejemplo el uso de metodologías como la de Desarrollo y Operaciones *(Development and Operations DevOps)* y la aplicación de herramientas de integración continua que reducirían la complejidad operatividad inherente. Así mismo, la definición de un número limitado pero efectivo de microservicios, mitigaría el radio de latencia existente en un sistema, al igual que el tiempo que tomaría sincronizar la información en cada uno de los microservicios. Complementario a las soluciones descritas, se nombran el uso de patrones para la gestión de fallos y estabilidad del sistema como el *Circuit Breaker* y el *Bulkhead* (Cerny et al., 2018), así como estrategias para la coordinación entre microservicios como lo son la *Coreografía* y la *Orquestación* (Cerny et al., 2018).

Es importante, además, destacar los beneficios adquiridos al incorporar arquitectura de microservicios en la incorporación de las *IP* y más aún en las *iPaaS*, según el análisis elaborado por (Nebel, 2019). Para ello, se describen los atributos de calidad sobre los cuales se ejecuta dicho análisis, empezando por el **Uso de recursos**, el mismo que se ve reflejado por una escala a demanda por parte de los microservicios requeridos, y no por toda la plataforma *IP* en sí. El impacto en este atributo es mayor si se incorpora el uso de plataformas de orquestación de contenedores sobre clústeres para un escalamiento elástico. En cuanto a **Capacidad**, la misma se beneficia de las bondades del atributo anterior, más aún si se utiliza

infraestructura en la nube, en donde el uso de recursos es a demanda sin ninguna limitante. Respecto a la **Mantenibilidad**, se presentan beneficios relacionados a la naturaleza propia en la segmentación e independencia entre los microservicios, tales como la **Modularidad** y la **Reusabilidad**. En cuanto a la **Instalabilidad**, la misma goza de un efecto positivo siempre y cuando existan recursos y metodologías que aporten a la misma, como es el caso de *DevOps*. Respecto a la **Tolerancia a fallos**, se describió que la misma es alcanzable con la aplicación de técnicas y patrones como los descritos en el párrafo anterior, solventando así los problemas arrastrados de sistemas distribuidos. Se puede indicar además que, la redundancia en hardware inherente de infraestructuras en la nube, permite una mejor tolerancia a defectos de hardware en sí. Paralelamente, los atributos de **Recuperabilidad** y de **Disponbilidad**, también se ven beneficiados de las mismas técnicas aplicadas en la tolerancia a fallos, para garantizar su efectividad. La **Reemplazabilidad**, tiene un efecto positivo con el uso de microservicios porque solo se requiere reemplazar aquellos que lo necesitan sin tener que desplegar el sistema entero.

Una de las propiedades que cuenta con un factor de impacto negativo en una arquitetura basada en microservicios es la **Seguridad**. Según las observaciones del autor, existe un mayor número de componentes o servicios expuestos que pueden comprometer la seguridad. Así mismo, la complejidad en la supervisión o monitoreo es proporcional a la cantidad de microservicios o partes expuestas. Por último, problemas en la gestión de credenciales de repositorios que pueden existir por el número de microservicios y defectos de seguridad introducidos por los equipos de desarrollo, pueden impactar negativamente a una plataforma. Sin embargo, la mitigación a estas deficiencias puede aplicarse siempre y cuando se incorporen enfoques de protección de fronteras, mediante el uso de los denominados *Edge Services*, y de soluciones en la orquestación de contenedores como *Kubernetes*, en donde existen componentes y funcionalidades de seguridad como por ejemplo el manejo de credenciales mediante *Secrets* que cifran los datos que sean almacenados en los mismos, o el manejo de control de accesos mediante objetos basado en roles y de usuarios.

Por último, uno de los atributos con mayor impacto de las *IP* basadas en microservicios es el **Perfil de usuario**. Según el análisis del autor, se cita que el cambio de quienes realizan tareas de integración "[...] permite mayor fluidez en las mismas, evitando costos innecesarios y optimizando recursos humanos. Así, ingenieros de sistemas pueden enfocarse en proyectos de integración compleja, delegando a usuarios menos técnicos tareas de integración simple como integrar sus datos con herramientas diarias de negocio, como el correo [...]“

Tabla 2.2: Aplicabilidad de *IP* basada en microservicios

Factores		Aplicabilidad de IP basada en microservicios
Despliegue de IP	**Plataforma de orq. de contenedores**	Beneficiado
	Ubicación de las IP	No beneficiado o No aplica
Particularidades de Sistemas a integrar	**Alta cantidad de sistemas**	Beneficiado
	Volatilidad o Inmadurez	Beneficiado
Particularidades de Datos a integrar	**Volumen de datos/accesos**	Beneficiado
	Complejidad de datos	Beneficiado
Cantidad de Organizaciones	**Escenario Interorganizacional**	Beneficiado
	Escenario Intraorganizacional	No beneficiado o No aplica

2.3.1. Tecnologías para el manejo de microservicios

Considerando la naturaleza de segmentación que se establece en las arquitecturas basadas en microservicios, puede llegar a existir un grado de complejidad proporcional a la cantidad de microservicios que definan a un sistema, dentro de esta arquitectura. La complejidad operacional surge cuando existen muchos servicios o puntos de un sistema que deben ser monitoreados para determinar su correcto funcionamiento, así como la ejecución de los procesos operativos para el escalamiento del sistema en sus partes más críticas. Así mismo, a mayor número de microservicios, mayor será la cantidad de puntos que deberán mantener actualizado su información; y por lo tanto, mayor la complejidad al momento de administrar cada uno de los repositorios independientes para el efecto (según dicta el lineamento que para cada microservicio deberá existir un repositorio independiente y exclusivo).

Con el afán de solventar la complejidad operacional presente en sistemas que manejan microservicios, existen una serie de tecnologías que favorecen tanto el desempeño, como la gestión misma del sistema, proporcionando capacidades como la escalabilidad, la alta disponibilidad, tolerancia a fallos, mantenibilidad, recuperabilidad, optimización de uso de recursos y la seguridad.

Docker y Kubernetes

Para comprender el uso de *Docker*, es necesario conocer lo que es un contenedor. Según (Shah and Dubaria, 2019), se entiende por contenedor a un ambiente aislado y cerrado, que agrupa en sí todos los recursos que una aplicación necesita para funcionar correctamente. Estos contenedores permiten que muchas aplicaciones corran dentro de un mismo ambiente o máquina y compartiendo recursos, a lo cual se lo ha denominado como **Contenerización**. *Docker* es la tecnología *open-source* detrás de la implementación y respectiva automatización de los contenedores, en diferentes ambientes de ejecución como *Linux* y *Windows*. La compañía que fomenta la tecnología de código abierto de *Docker*, lleva el mismo nombre, *Docker Inc*. Si bien no es la única solución dentro del mercado, para la generación de contenedores, *Docker* ha crecido con el pasar de los años para convertirse en el formato predeterminado y de fácil uso para dicha labor. Ahora bien, la generación de los contenedores, usando *Docker*, se resume en: i) partir de un archivo denominado *Dockerfile*, con la definición de como construir el archivo binario de lo que tendrá un contenedor. A este archivo se lo conoce como imagen *(Image)*. ii) Se ejecutan una serie de comandos para empaquetar la aplicación y generar la imagen, a partir del *Dockerfile* y subirla a un espacio público o privado, denominado Registro de contenedores *(Container Registry)*. iii) Estas imágenes pueden ser referenciadas para su uso y ejecución en plataformas que admiten contenedores como *Kubernetes*, *Docker Swam*, *Mesos* o *HashiCorp Nomad*.

En cuanto a la orquestación de contenedores, el más conocido y de gran renombre en la industria es *Kubernetes*. Es una plataforma de código abierto que orquesta la ejecución de uno o varios contenedores, en un entorno clusterizado. Dentro de su gestión de contenedores, se identifican responsabilidades como el despliegue de contenedores, escalamiento horizontal, y el balanceo de la carga (Shah and Dubaria, 2019). Las ventajas de un orquestador de contenedores, como lo es *Kubernetes*, van desde la i) automatización de operaciones con una sola interfaz *API*, ii) abstracción de la infraestructura, iii) monitoreo o supervisión del estado de los servicios y iv) el favorecer con la implementación de flujos o *pipelines* para la integración y el despliegue continuo (Campbell).

Es importante mencionar que *Kubernetes* no es una plataforma como servicio *(Platform as a Service PaaS)*. Crear y administrar un clúster de *Kubernetes* puede ser un tanto complejo, debido a los diferentes componentes y su interacción que requieren ser configurados y monitoreados. Por esa razón, existen proveedores que ofrecen esta plataforma para abstraer toda esa complejidad y facilitar la incorporación de la misma dentro de la infraestructura de los clientes. En la actualidad, se cuentan con diferentes proveedores de servicios de cómputo en la nube que ofrecen a *Kubernetes* como un servicio, por ejemplo: *Amazon Elastic Kubernetes*

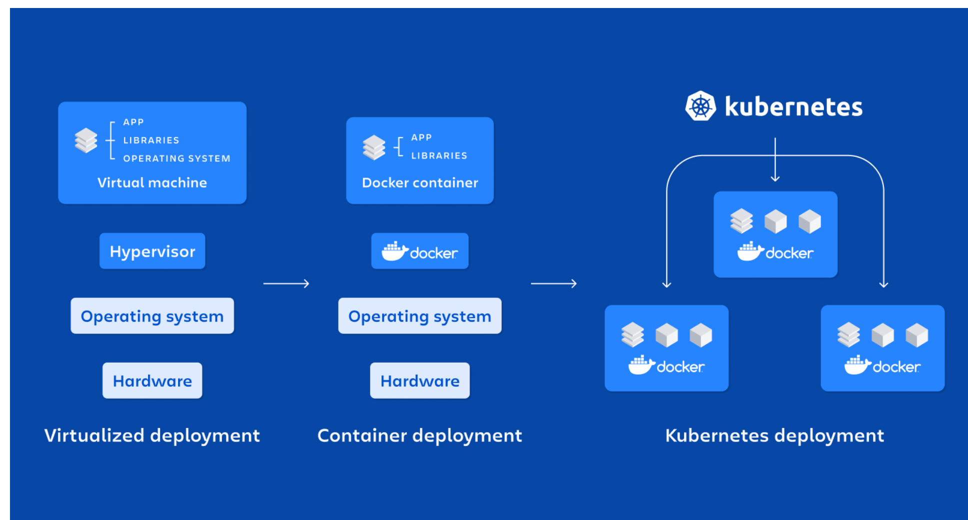


Figura 2.3: Estructura de una máquina virtual, un contenedor y un orquestador de contenedores (Campbell)

Services (EKS) (Services), *Azure Kubernetes Service (AKS)* (Microsoft), *Google Kubernetes Engine (GKE)* (Cloud, a), entre otros.

2.4. Criterios de Seguridad y/o Privacidad de la información

Los datos son el motor principal que define la existencia y el subsecuente funcionamiento de las *iPaaS*. En la actualidad, la diversidad y volumen de datos generados por diversas fuentes, sean estos usuarios, dispositivos *IoT* o sistemas, ha crecido exponencialmente frente a los años anteriores, según se ha pronosticado en (Balasubramanian et al., 2021). Sin embargo, a mayor volumen de datos, mayores son los riesgos de que dicha información sea vulnerada, a nivel de filtraciones, alteraciones o falsificaciones. Un sistema que consuma y procese datos críticos como información del área de la salud por ejemplo, o datos de información bancaria o de ubicación personal, y que no apliquen un cuidado especial en el tratamiento de esta información, podría generar graves perjuicios a las personas dueñas de la misma, durante el robo o alteración sin consentimiento, en beneficio de un actor malicioso.

Según el reporte de (IBM, 2022), el costo producido a nivel mundial debido a las brechas de seguridad en los datos consumidos y procesados por las industrias de todos los ámbitos ha sido de alrededor de 4.5 millones de dólares, un 2.6 % mayor al del año 2021. Las áreas de la Salud, Finanzas,

Tabla 2.3: TOP 5 en costos por brechas de seguridad 2022-2021

Sector industrial	**2021($Millones$)**	**2022($Millones$)**
Salud	$10.10	$9.23
Financiero	$5.97	$5.72
Farmacéuticas	$5.01	$5.04
Tecnología	$4.97	$4.88
Energía	$4.72	$4.65

Farmacéuticas, Tecnología y de Energía, ocupan los 5 primeros puestos de los sectores donde los costos por brechas de seguridad en los datos, ha sido importante.

Las causas que han desembocado en la pérdida sobre la seguridad de los datos en la industria son variadas. Respecto a la infraestructura, políticas de cero confianza recomendadas para mitigar las vulnerabilidades no fueron adoptadas correctamente y esto en consecuencia, ha provocado que brechas de seguridad como el error humano, ocupen el 22 % del total de incidencias de seguridad. Le siguen fallas en la gestión de Tecnologías de la información *Information Technology IT* con un 25 %. Un 12 % lo ocupa ataques conocidos como *Ransomware*, en donde existe un cifrado intencional y no autorizado sobre los datos a cambio de un valor monetario por su liberación.

En consecuencia, la necesidad por incorporar mecanismos de seguridad sobre los diferentes niveles y recursos de una organización es imperativo. Si hablamos de plataformas *iPaaS*, sean que estas corran en un ambiente de cómputo público o privado en la nube, o en infraestructura local, los criterios de seguridad sobre la información toman un papel relevante para determinar su factibilidad de uso. Artículos como (Neifer et al., 2021), ponen en evidencia la importancia en el cuidado con los datos. Dentro del análisis llevado por sus autores, como parte de un estudio empírico cualitativo, basado en un conjunto de entrevistas, a un grupo de expertos de los diferentes proveedores de servicios de software, se destaca a la protección y seguridad de los datos, sobre todo para Europa con la Regulación General de Protección de Datos *(General Data Protection Regulation GDPR)* Commission, como requisitos fundamentales que todos los proveedores de servicios *iPaaS* deberán cumplir. Si bien estas normas garantizan el cuidado con el manejo de los datos de los clientes en general, su aplicabilidad puede resultar problemática en áreas como la usabilidad y la experiencia de usuario, en comparación con sus similes en América, en donde la aplicación de la normativa no es tan rigurosa al final del día. Es así que el factor ubicación juega un papel preponderante a la hora de elegir la infraestructura sobre la cual levantar un *iPaaS*.

2.5. Infraestructura como código

Se conoce a la Infraestructura como código *Infrastructure as a Code IaC* como un nuevo alcance en la automatización del levantamiento de infraestructura, con bases en el desarrollo de software, con el objetivo de adquirir la gobernanza necesaria en los procesos y recursos que la definen. Entre los beneficios que se obtienen en la adopción de este paradigma de gestión, se resumen en i) creación de sistemas confiables, seguros y de costos efectivos, ii) reducción del esfuerzo y los riesgos asociados a cambios de infraestructura (muchas de las veces manuales), iii) habilitación de la infraestructura con los recursos que se requieren, en el tiempo que se lo necesita, iv) mejora en la velocidad ante la resolución de fallas, v) uso de infraestructura de *IT* como habilitador en una entrega rápida de valor (Morris, 2020).

Tradicionalmente, toda infraestructura de *IT* era provisionada manualmente con el uso de ciertas herramientas. Con el advenimiento de la computación en la nube, el aprovisionamiento pasó de ser algo complejo y propenso a errores, a ser sencillo y confiable de ejecutar, sobre todo por la abstracción que muchos de los proveedores en la nube, incorporaron sobre las tareas de configuración complejas a la hora de levantar un servidor o grupo de servidores, almacenamientos, redes, etc. Se complementa a esta propuesta, el hecho de que los mismos proveedores han expuesto la gestión de su infraestructura a través de servicios *APIs*, abriendo la posibilidad de disponibilizar una serie de herramientas que faciliten la incorporación de este nuevo paradigma de gestión de infraestructura en las organizaciones. Así mismo, la disponibilidad de sistemas de control de versiones *(Version Control System VCS)* como *GIT*, permiten respaldar los cambios de infraestructura declarados en estos archivos y la disposición de flujos automatizados sobre estos archivos para el levantamiento de infraestructura en los proveedores de cómputo en la nube (Véase Figura 2.4).

Las herramientas que en la actualidad ayudan en la incorporación de *IaC*, se encuetran catalogadas en dos grupos, las herramientas que provisionan la infraestructura como por ejemplo *Terraform, Pulumi, Cloudformation*; y las herramientas que administran la configuración de las infraestructuras como por ejemplo *Ansible, Chef, Puppet.*

2.5.1. Terraform

De entre las herramientas más conocidas en el mercado tecnológico para dar soporte a modelos *IaC*, sobresalen dos con capacidades similares, pero de diferente enfoque a la hora de esquematizar las definiciones de infraestructura requerida. Por un lado tenemos a *Terraform*, una herramienta *open-source* creada por la empresa HashiCorp y que basa la definición de la infraestructura a través de un lenguaje de configuración de alto nivel llamado

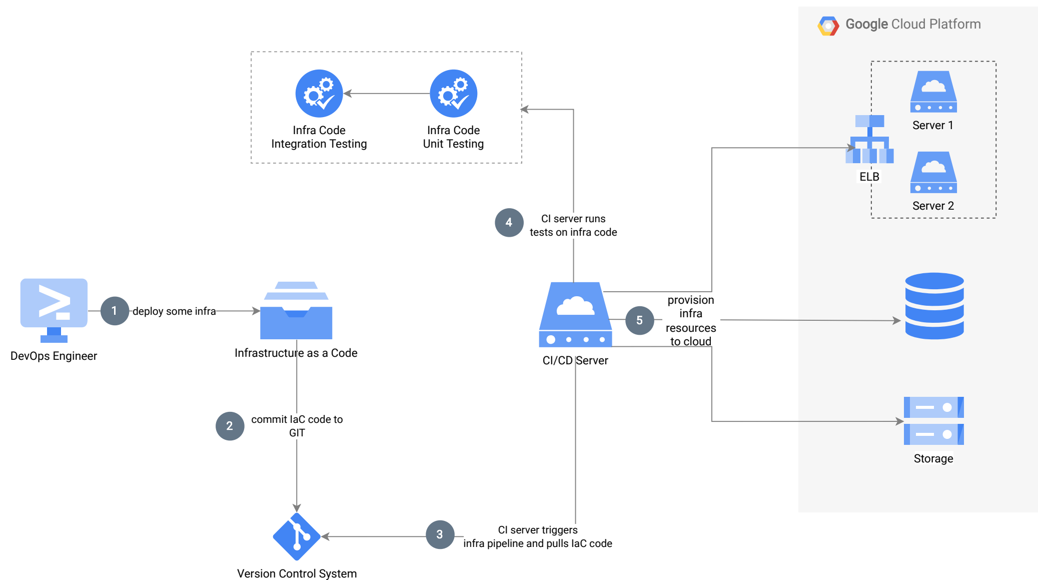


Figura 2.4: Implementación de *IaC* con VCS, pruebas y despliegue en nube

Hashicorp Configuration Language (HCL) (Howard, 2022). El flujo básico de ejecución que maneja Terraform (Véase Figura 2.5), se descompone en tres etapas i)La **escritura**, que se centra en el desarrollo del código requerido para el levantamiento de la infraestructura, y definidos en un archivo de extensión *tf*. ii)El **plan**, que es la etapa en la cual se genera los pasos a ejecutar en base a la configuración definida en los archivos *tf* y con el uso de herramientas como interfaz de línea de comandos *(Command Line Interface CLI)* o mediante una interfaz de lenguaje de alto nivel que se comunique con el marco de trabajo de *Terraform* vía *Cloud Development Kit CDK*. Esta serie de pasos definen acciones de creación, actualización y eliminación de los recursos de la infraestructura deseado, devolviendo como resultado un archivo denominado *state*. Este archivo será usado posteriormente por *Terraform* para garantizar el concepto de *Idempotent* sobre la infraestructura creada. iii) La **aplicación**, es la última etapa en la que se levanta la infraestructura deseada mediante la ejecución del plan contra el correspondiente proveedor de cómputo en la nube.

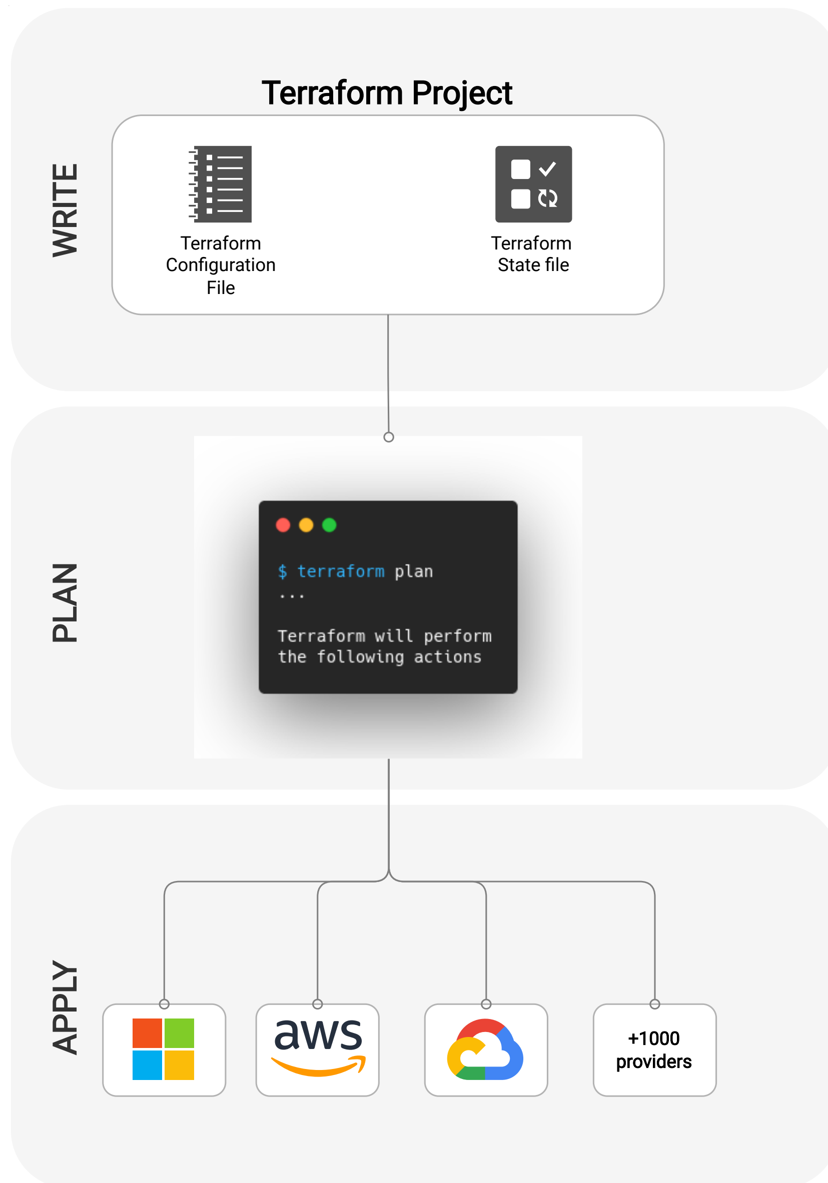


Figura 2.5: Flujo de ejecución de alto nivel de *Terraform*

2.5.2. Pulumi

Otra herramienta de aprovisionamiento de infraestructura es *Pulumi*, que al igual que su símil, *Terraform*, también ofrece capacidades para administración bajo el modelo *open-source*. Sin embargo, la diferencia entre ambos radica en que *Pulumi* no utiliza lenguaje de dominio específico para esquematizar la infraestructura, y en su lugar, permite el uso de lenguajes

demo2 > ! Pulumi.yaml > YAML > {} resources > {} fn > {} properties > runime

```
20    fn:
21      type: aws:lambda:Function
22      properties:
23        runime: python3.8
24        handler: handler.handler
25        role: ${role.arn}
26        code:
27          fn::fileArchive: ./function
28    # A REST API to route requests to HTML content and the Lambda function
29    api:
30      type: aws-apigateway:RestAPI
31      properties:
32        routes:
33          - path: /
34            localPath: ww
35          - path: /date
36            method: GET
37            eventHandler:
38  outputs:
39    # The URL at which the REST
40    url: ${api.url}
41
```

Resource: aws-apigateway:index:RestAPI

The RestAPI component offers a simple interface for creating a fully functional API Gateway REST API. The REST API can define any number of routes, each of which maps a path and HTTP method to one of (1) an event hander route that invokes a Lambda Function (2) a local path route which uploads local files into an S3 bucket and serves them or (3) an integration target such as an HTTP proxy or service integration.

Inputs

apiKeySource

Figura 2.6: Retroalimentación enriquecida en *IDE*, en la definición de *IaC* con *Pulumi*

de propósito general como *Python, Javascript, Go, Java, .NET* y hasta lenguaje de marcado en formato *YAML*, para su definición. Si bien en el pasado, esta característica era la principal ventaja de esta herramienta sobre su competidora, que sufría de una falta de poder y expresividad necesaria en la definición del despliegue de infraestructuras complejas, según se indica en (Campbell, 2020); a día de hoy, ya no es un diferenciador más, gracias en parte, a la disponibilidad del *CDK* de *Terraform*, con soporte para uso desde los principales lenguajes de programación existentes en la actualidad (Howard, 2022). No obstante, la capacidad de *Pulumi* de enriquecer los entornos de desarrollo integrales *Integrated Development Environment IDE*, junto con el lenguaje de programación favorito, es superior a lo que *Terraform* ofrece actualmente, con características como completar código, tipado fuerte, rica documentación de los recursos, resaltado de errores, entre otros. Véase Figura 2.6.

Es importante destacar que en la actualidad, no existe una propuesta o investigación que involucre el uso de modelos de automatización del tipo *IaC*, para el levantamiento de plataformas *iPaaS* en las organizaciones. Por lo tanto, existe un campo abierto para la puesta en marcha de análisis y ejecución de soluciones que incorporen estos modelos para fortalecer el

ciclo de vida de estas plataformas en las organizaciones; más aún, cuando la evolución de las tecnologías y de los requerimientos del mercado siguen en continua aceleración.

Capítulo 3

Desarrollo de la Solución

En éste capítulo se presenta el desarrollo de la propuesta planteada a la problemática descrita en el Capítulo 1. El capítulo describe con el detalle necesario, los distintos componentes a ser desarrollados, así como los recursos pre existentes y necesarios para el montaje de la *iPaaS*. El contenido está organizado de la siguiente manera: en la sección 3.1, se expondrá el diagrama de Arquitectura lógica de la solución de *iPaaS*, junto con la descripción de los elementos que lo componen. Se complementa la descripción y funcionamiento de los componentes con la presentación de diagramas de Lenguaje de Modelado Unificado *Unified Modeling Language UML*. Seguido de ello, en la sección 3.2, se presentará el esquema de datos requerido para el soporte de la autogestión en el levantamiento de los servicios de Integración. A continuación, en la sección 3.3, se expone el Diagrama de arquitectura tecnológico con la propuesta en el uso de soluciones en la nube. Por último, en la sección 3.4, se presentan los recursos necesarios para su despliegue, mediante el uso de herramientas de tipo *IaC*.

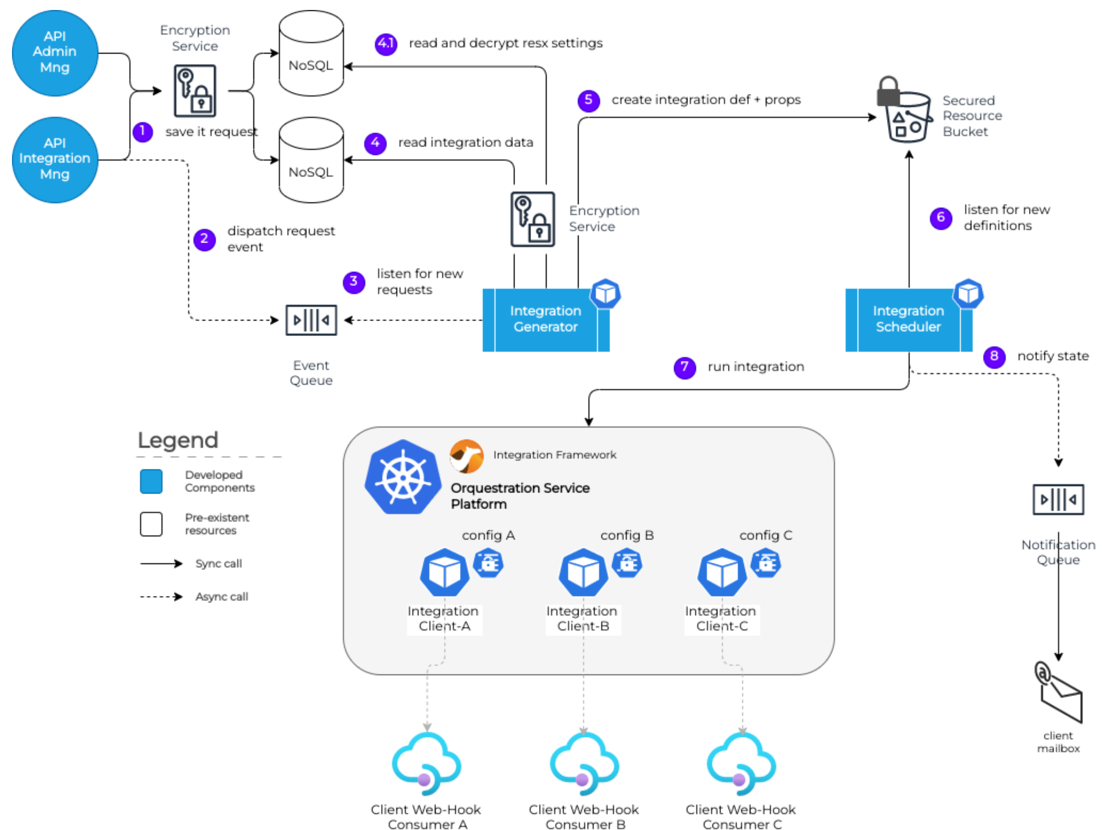


Figura 3.1: Arquitectura lógica de in *iPaaS* autogestionable

3.1. Diseño de arquitectura lógica

Con el objetivo de alcanzar la autogestión de los servicios de integración y su ejecución dentro de la plataforma *iPaaS* para el efecto, se contempló el uso de una arquitectura basada en microservicios. La justificación para dicha elección radicó en la búsqueda de flexibilidad a la hora de escalar los diferentes componentes y/o servicios de cara al volumen de transacciones que la plataforma espera procesar dentro de sus operaciones.

Por otro lado, y basándose en la investigación llevada a cabo en el Capítulo 2 sobre *frameworks* de integración, se eligió a la solución *open-source* de *Apache Camel K* como el núcleo principal de esta propuesta de *iPaaS*. Se justifica su elección por las siguientes razones:

i) Es un *framework* de integración maduro con un amplio soporte para un abanico de tecnologías y/o herramientas con las cuales poder acoplarse, tanto en infraestructuras locales, como con proveedores de infraestructura en la nube.

ii) La solución fue diseñada para correr dentro de plataformas de orquestación como *Kubernetes*, permitiendo que los servicios integradores desarrollados bajo este *framework* puedan gozar de las capacidades que la plataforma subyacente permite, por ejemplo: escalabilidad y redundancia.

iii) Si bien los lenguajes sobre los cuales se pueden escribir las integraciones varían desde *XML, YAML, Groovy, Kotlin, o Javascript*, el core o el tiempo de ejecución *(runtime)* sigue siendo Java y gracias a la incorporación de la plataforma *Quarkus*, existe una optimización al momento de ejecutar procesos *Java* con tiempos de ejecución rápidos y con bajo consumo de memoria.

De esta manera, la arquitectura que se diseñó para abordar la autogestión de las integraciones, por parte de los clientes, contempla tanto los recursos de terceros y de tipo *open-source*, así como servicios desarrollados. El principal enfoque que se dió a esta propuesta de arquitectura yace sobre los servicios desarrollados, puesto que los recursos de terceros componen la lista de prerequisitos que deben ser atendidos y disponibilizados por la empresa adquiriente de la solución *iPaaS* para el funcionamiento de la solución. A continuación, se describen los componentes desarrollados dentro de la arquitectura planteada.

3.1.1. API Admin & API Integration Management

El primer grupo de componentes que se diseñaron fueron los servicios de gestión de la información pertinente a las Integraciones y los clientes dueños de las mismas. Según se aprecia en la Figura 4.5, los servicios de gestión se diseñaron a partir de un grupo común de capas que engloban:

i) El punto de acceso o capa de presentación de tipo *API REST*, para interactuar con el servicio.

ii) Una capa de servicios para definir un conjunto de funcionalidades, entre ellos el cifrado de datos sensitivos, y un servicio para la notificación asíncrona de cualquier tipo de evento relacionado al ciclo de vida de una integración, para su posterior tratamiento por parte de otros componentes de la plataforma.

iii) Se complementó a este diseño la incorporación de una capa de acceso a las fuentes o repositorios en donde se almacenará la información pertinente.

Cabe indicar que por la naturaleza de la información a tratar, se planteó el uso de soluciones de repositorios de datos no relacionales *NoSQL*, ganando de esta manera la flexibilidad necesaria dentro de la evolución de la solución a nuevos requerimientos.

Los diagramas de secuencia presentados en las Figuras 3.3 y 3.4 detallan los pasos de ejecución de distintas tareas llevadas a cabo por los servicios y recursos participantes durante la gestión de las integraciones.

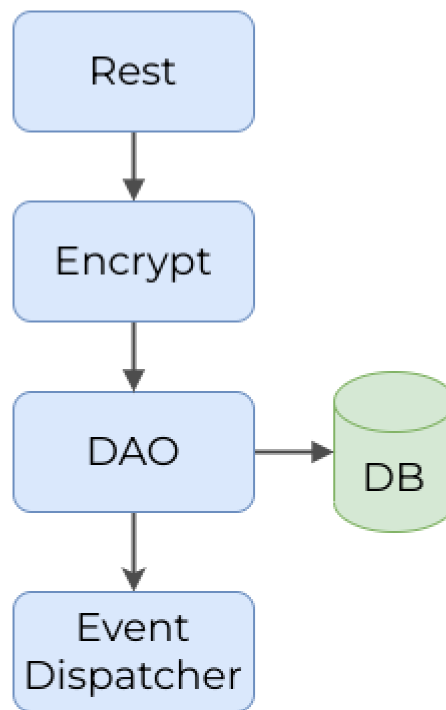


Figura 3.2: Diagrama de bloques de servicios *APIs*

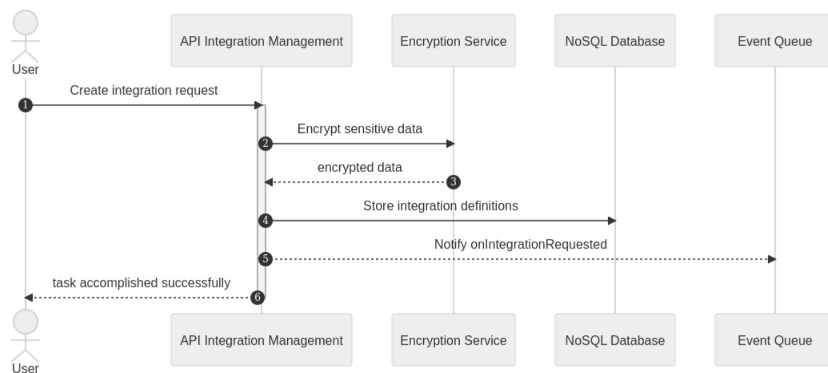


Figura 3.3: Diagrama de secuencia de servicio *API* Integration Management

3.1.2. Integration Generator

Se procedió con el diseño de este servicio con el objetivo de centralizar la interpretación de los datos registrados a través de los servicios *APIs* y la posterior creación de las definiciones requeridas por el *framework* de

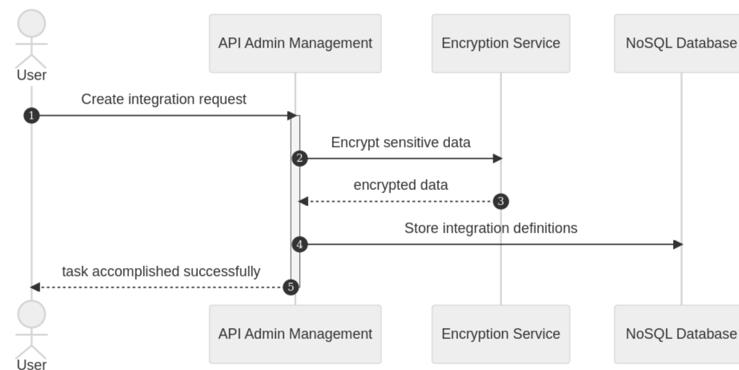


Figura 3.4: Diagrama de secuencia de servicio *API* Admin Management

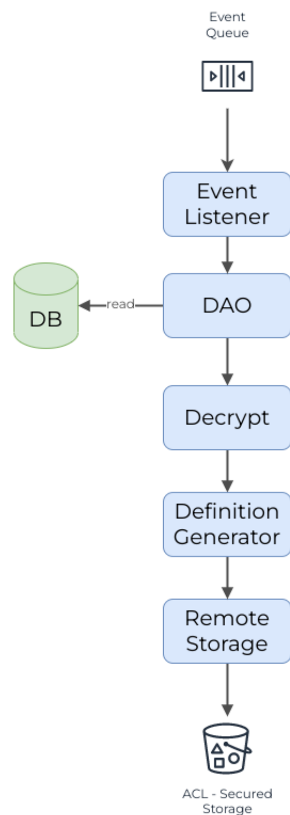


Figura 3.5: Diagrama de bloques del servicio Integration Generator

integración elegido para su ejecución. Las tareas de este servicio se detallan a continuación:

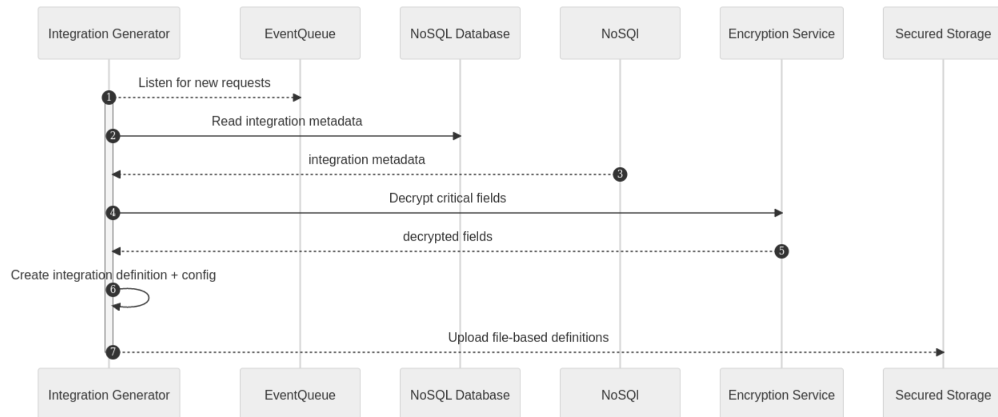


Figura 3.6: Diagrama de secuencia de servicio Integration Generator

i) Se conecta a un sistema de colas para la recepción y/o escucha de mensajes que representan los eventos asociados a las tareas de gestión en el ciclo de vida de las integraciones, provenientes de servicios de nivel superior.

ii) A partir de esa información, se ejecutan consultas al repositorio de datos principal.

iii) Luego se ejecuta procesos de desencripción de los datos críticos consultados.

iv) Y, posterior a ello, se realiza la generación de las definiciones requeridas por el *framework* de integración de la plataforma, para ejecutarlos dentro de su ambiente.

v) Al final del proceso, estas definiciones son cargadas en un sistema de almacenamiento de objetos para la consecuente descarga y ejecución de parte de otros servicios de la plataforma.

Servicios de almacenamiento de tipo *SaaS* como *Amazon Web Service Simple Storage Service (S3)* o *Google Cloud Storage (GCS)* favorecen a fortalecer la solución debido a sus costos moderados y sobre todo, con la granularidad esperada en el control de accesos a la información que ahí resida. Este punto es importante, puesto que las definiciones podrían contener información crítica como credenciales (datos de usuarios y contraseñas) para el acceso a servicios tanto internos, como externos en el consumo y publicación de información.

Para incorporar aspectos de seguridad de la información, se estableció el uso de componentes de *Kubernetes* denominados *Secrets*, para registrar datos de configuración de las integraciones, y que las mismas no sean explotadas para fines contrarios al uso de la plataforma.

En la Figura 3.6 se puede apreciar la secuencia de tareas que el servicio

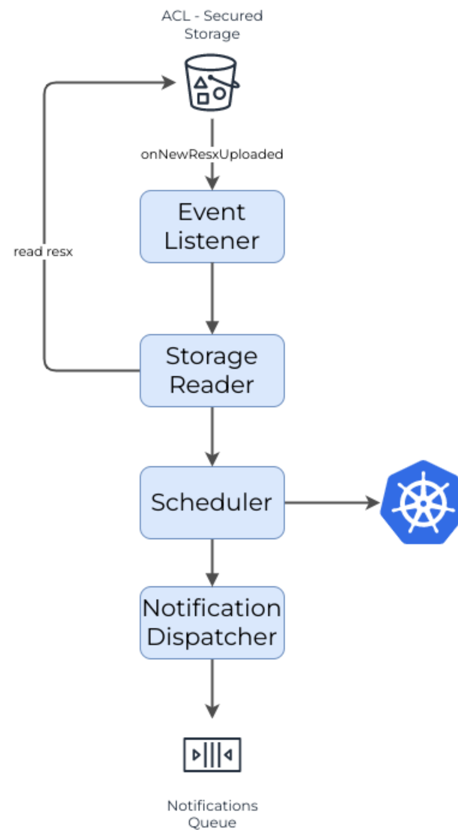


Figura 3.7: Diagrama de bloques del servicio Integration Scheduler

ejecuta para cumplir con su función primaria.

3.1.3. Integration Scheduler

Se elaboró el diseño de este servicio con el objetivo de poder ejecutar las integraciones de los clientes, dentro de un ambiente de orquestación de servicios como *Kubernetes*, utilizando las definiciones que fueron construidas por el *Integration Generator* y que se encontrarán almacenadas en el repositorio para el efecto.

i) Según la Figura 3.7, el servicio consta de un componente que escucha eventos relacionados a cargas de objetos en el almacenamiento remoto designado.

ii) A partir de ahí, se ejecuta la lectura de dichos objetos para luego planificar su ejecución dentro de la plataforma de orquestación de servicios elegido.

ii) Por último, notificará el estado final de la tarea principal, a través de un despachador de eventos y el sistema de colas respectivo, permitiendo que,

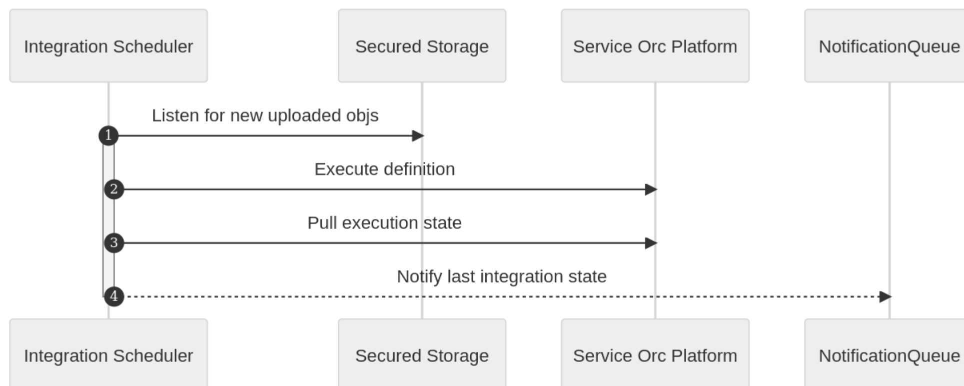


Figura 3.8: Diagrama de secuencia de servicio Integration Generator

otros servicios de la plataforma puedan ejecutar tareas de cierre del proceso principal, como por ejemplo, notificación mediante correo electrónico hacia el cliente sobre la ejecución exitosa o fallida del servicio de integración.

En la Figura 3.8 se puede apreciar la secuencia de tareas que el servicio ejecuta para cumplir con su función primaria.

3.2. Repositorio y esquema de datos

Para el diseño del esquema de datos necesario de la plataforma, se llegó a adoptar un enfoque no relacional, o lo que comúnmente se lo denomina soluciones *NoSQL*. La razón de tal elección estuvo sustentada en algunos aspectos, entre los más importantes, la flexibilidad en el cambio evolutivo de las estructuras de las entidades que representarían al modelo de negocio sobre el que trabajará el *iPaaS*. De esta manera, nuevas funcionalidades o requisitos para la plataforma *iPaaS*, serían incorporados fácilmente y sin comprometer su estabilidad.

Ahora bien, para incorporar este enfoque en el manejo de la información, se eligió al producto de base de datos no relacional *MongoDB*, cuya versión comunitaria es *open-source* y nos permite su utilización sin requerir de licencia alguna. Adicional a este repositorio, existen soluciones de base de datos NoSQL de tipo *SaaS* que abstraen el esfuerzo necesario para su administración y optimiza el enfoque hacia la incorporación de la solución en sí. Soluciones como las de *Amazon Web Service Dynamo* o las de *Google Cloud Firestore*, permiten su utilización a un coste bastante flexible; más aún, cuando la tasa de transacciones sobre las mismas son bajas, se puede aprovechar sin problema la capa gratuita que cada una expone para dichas

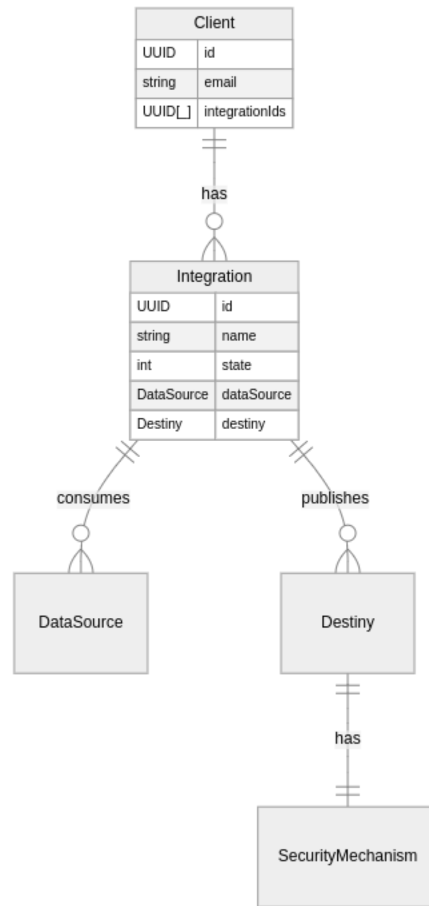


Figura 3.9: Diagrama Entidad Relación del modelo de datos

capacidades.

Considerando la elección de *MongoDB* como el repositorio principal para la plataforma *iPaaS*, se llegó a diseñar el esquema de datos que se presenta en la Figura 3.9. Para el caso de las entidades *Integration*, *Destiny* y *SecurityMechanism*, se definió la utilización del concepto de Herencia de la programación orientada a objetos, una funcionalidad soportada por base de datos documentales, para albergar diferentes variaciones de cada una de las entidades, según corresponda. En la Figura 3.10 se definen los modelos del dominio y sus relaciones, así como las variaciones que pueden existir al momento de definir una fuente de datos, un destino y un mecanismo de seguridad al mismo. Para el caso puntual, se definió como fuentes de datos a un sistema de cola de mensajes llamado *RabbitMQ* y como destino, la exposición de un servicio *API REST* del lado del cliente, para la recepción

de la información, a modo de un *WebHook*. Por último, el mecanismo de seguridad aplicado para el manejo del destino de datos, se basa en el manejo de llaves o lo que se denomina como *API Key*, el cual se lo envía en cada solicitud, a través de la cabecera por defecto del mensaje llamada *Authorization*.

Se justifica entonces el uso de un almacenamiento no relacional, por el uso de una propuesta de repositorio a nivel documental, permitiendo de esta manera aprovechar conceptos de herencia sobre estructuras de datos que pueden variar en función de los requisitos de los clientes sobre el consumo de información. Por ejemplo, se podría representar como fuente de datos otro sistema de colas en la nube como *Google Cloud PubSub* (Refiérase a (Cloud, b) para mayor información), en donde las configuraciones de acceso y consumo difieren de las planteadas originalmente. Otro caso podría ser en la que el cliente decida consumir su información a través de un servicio web tradicional de tipo *SOAP* y cuyo mecanismo de seguridad difiere completamente del planteado en un inicio.

3.3. Diseño de Arquitectura tecnológica

Partiendo del diseño lógico presentado al inicio de este capítulo, se resolvió la presentación de dos tipos de propuesta de Arquitectura a nivel tecnológico para la plataforma *iPaaS*, tomando en cuenta dos enfoques. El primero es un enfoque en sitio, es decir, el montaje de la plataforma dentro de la infraestructura local de la empresa; y el segundo enfoque, orientado a una solución en la nube, con recursos auto-administrados o lo que se denominan *Serverless*, usando para ello, la nube del proveedor de *Google Cloud*.

3.3.1. Arquitectura tecnológica en infraestructura local

Según la Tabla 3.1, se definió las tecnologías y herramientas necesarias para la incorporación de la solución *iPaaS*, dentro de la infraestructura local de la empresa que lo requiera.

Es importante mencionar, que gracias al concepto y tecnología de contenerización de servicios, muchas de estas herramientas pueden ser instaladas y ejecutadas dentro de ambiente de orquestación de contenedores como lo es *Kubernetes*. Por lo tanto, si bien, su implementación no corresponde a la intención de esta tesis, se puede afirmar que soluciones como las *RabbitMQ*, *MongoDB*, *MinIO* y el conjunto *ELK*, pueden ser levantados dentro de la plataforma *Kubernetes*, con la asignación de recursos de cómputo adecuados para tal efecto.

Por último, en relación a los servicios a ser desarrollados como parte de la prueba de concepto y que forman parte de esta tesis, se definió el uso de tecnologías como *Python* y *FastAPI*. Sin embargo, y gracias al uso de una arquitectura basada en microservicios, no se descarta la posibilidad de

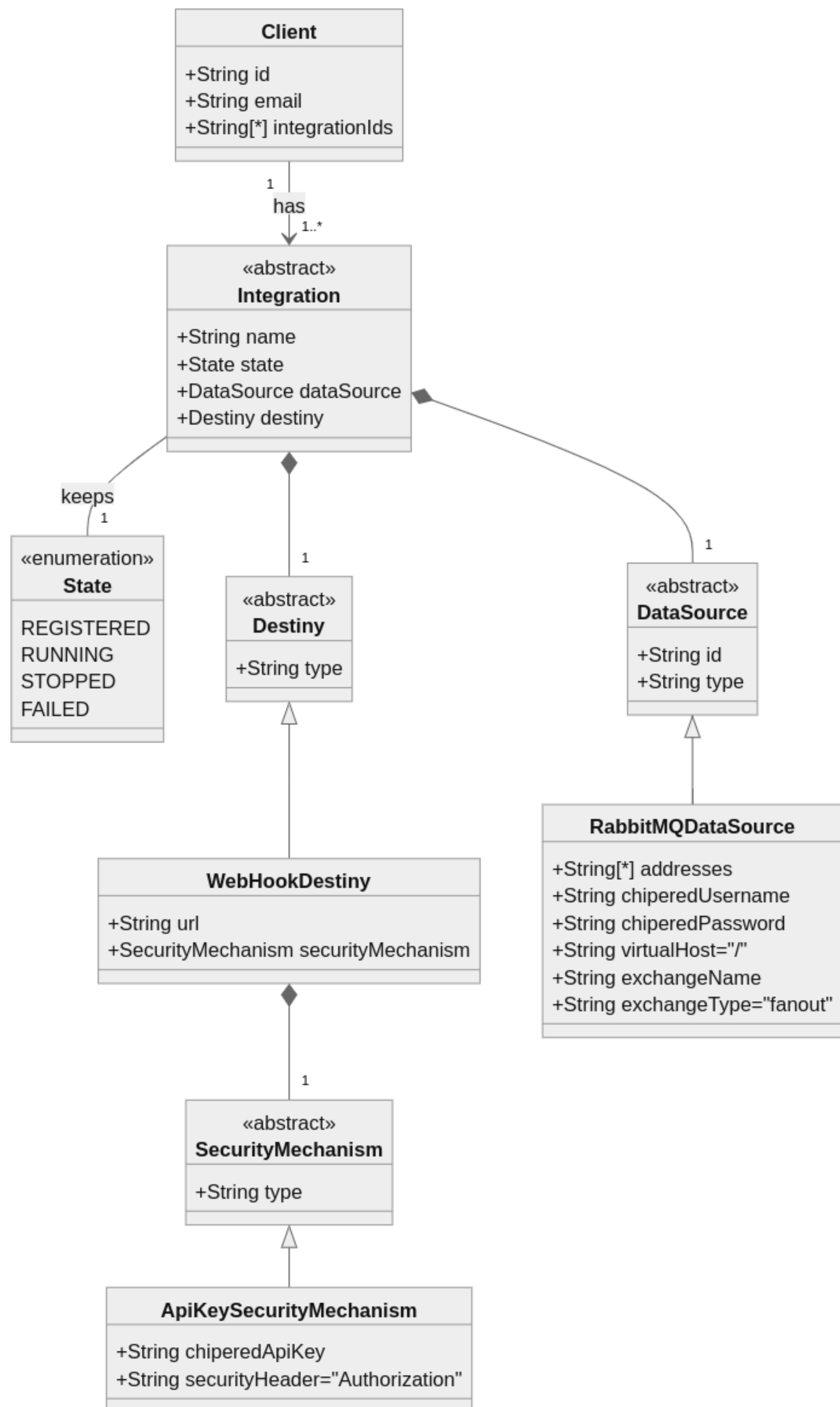


Figura 3.10: Diagrama de clases del dominio

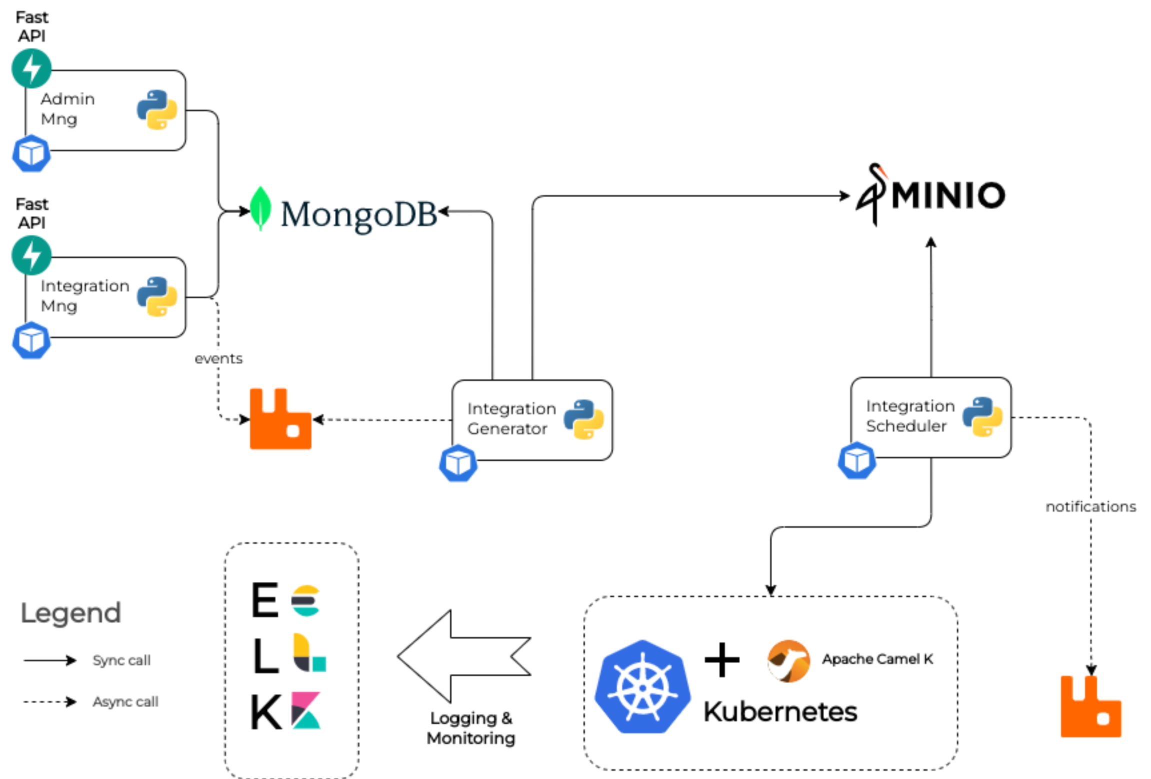


Figura 3.11: Arquitectura tecnológica en infraestructura local

Tabla 3.1: Recursos *open-source* de plataforma *iPaaS*

Herramienta	**Descripción**
RabbitMq	Sistema de cola de mensajes basado en protocolo AMQP y memoria para optimizar tiempo de operación
MongoDB	Base de datos no relacional basada en documentos. Maneja una versión comunitaria y de libre uso y otra bajo licencia.
ELK	Son los acrónimos de *Elasticsearch-LogStash-Kibana*, un conjunto de herramientas para captura, consolidación, búsqueda y monitoreo de logs.
Kubernetes	Orquestador de contenedores de uso libre
Apache Camel K	Marco de ejecución para sistemas de tipo *EIP*.
Minio	Solución de almacenamiento de objetos de alto rendimiento y de uso libre, con soporte a todas las capacidades claves que definen al servicio de *AWS S3*

usar otros lenguajes de programación y herramientas que encajen con los requerimientos o políticas definidas por una empresa.

3.3.2. Arquitectura tecnológica en la nube

Considerando el proveedor de cómputo en la nube, de la cual Location World mantiene su infraestructura, y con la idea de abstraer el esfuerzo de mantenimiento de los servicios que componen al *iPaaS*, se procedió a diseñar la siguiente propuesta de arquitectura tecnológica, usando los servicios e infraestructura que son provistos por *Google Cloud Platform*. En la Figura 3.12, se puede apreciar la arquitectura propuesta con los siguientes componentes claves:

i) Un componente de tipo *API Gateway* para la centralización de uno o varios microservicios expuestos por interfaz *API REST*.

ii) Un par de microservicios expuestos por interfaz *API REST*, que manejan los criterios de administración a alto nivel y la gestión enfocada que realizará un cliente final.

iii) Un componente *Serverless* denominado *KMS*, para la administración de procesos de cifrado.

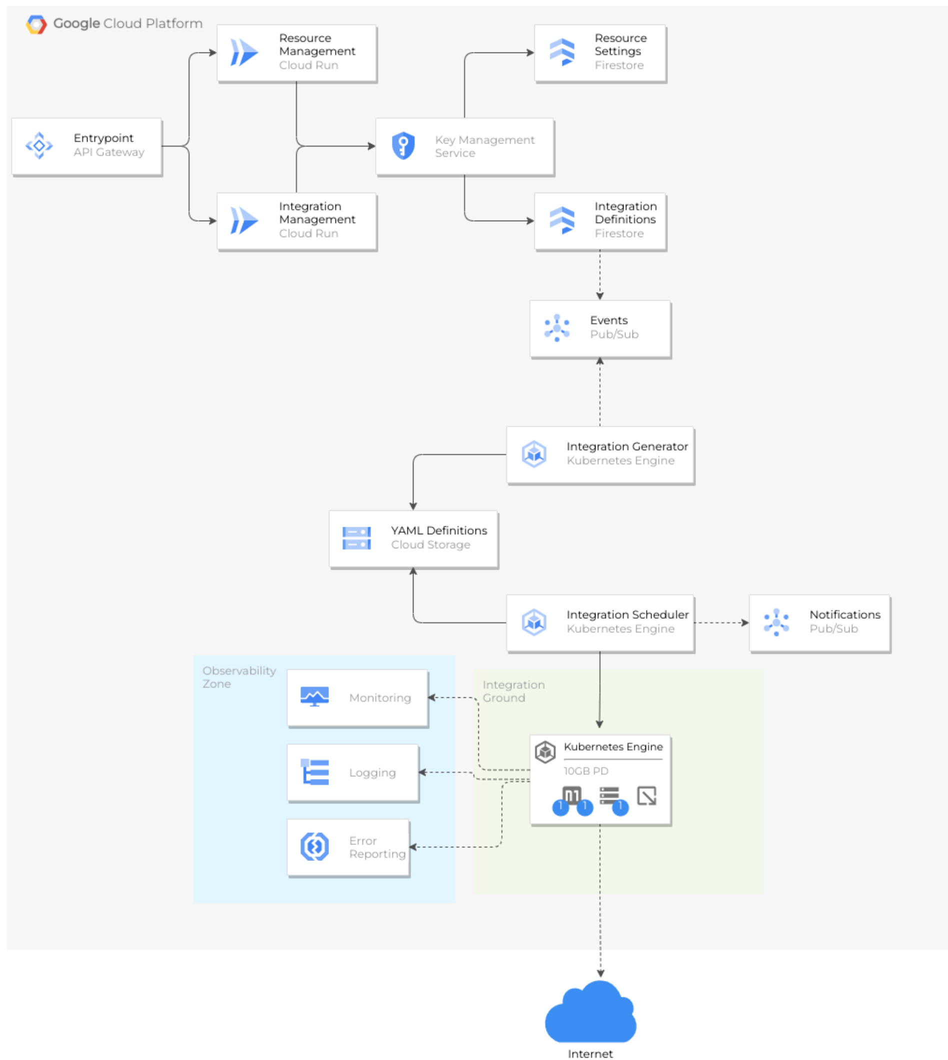


Figura 3.12: Arquitectura tecnológica con infraestructura en la nube

iv) Uso de un repositorio *NoSQL* y un sistema de colas completamente auto-administrables en la nube.

v) El uso del servicio en la nube para almacenamiento seguro de las definiciones de integración, a través de *Google Cloud Storage*.

vi) Utilización de componentes auto-administrables para la centralización y monitoreo de incidencias o logs generados en plataforma.

vii) Uso de una solución de *Kubernetes* en la nube para la carga de los componentes principales y las integraciones desplegadas por los clientes finales.

3.4. Personalización de *Apache Camel K*

Como parte de la incorporación del *framework* para desarrollo de integraciones como lo es *Apache Camel*, se presentó la necesidad de crear componentes personalizados que permitan establecer el flujo de integración desde una fuente de datos representada por una sistema de cola *RabbitMq* y un destino, representado por un servicio de tipo *API Rest* del sistema del cliente para la recepción de la información.

Gracias al soporte del Lenguaje de Especificación de Dominio *(Domain Specification Language DSL)* por parte de Apache Camel, se procedió a la definición del componente o modelo base sobre el cual se construirán todas las definiciones de las integraciones. La Figura 3.13 presenta la estructura del componente personalizado.

3.5. Infraestructura del *iPaaS* en la nube

Basándose en los recursos de software necesarios y detallados en secciones anteriores, al igual que los componentes indispensables para la autogestión de las integraciones y por último, la infraestructura subyacente en donde correrán todos ellos, es evidente la complejidad presente a la hora de ejecutar las tareas de instalación y posterior configuración de la plataforma de *iPaaS* y su posterior mantenimiento en el tiempo.

Por otro lado, disponibilizar la misma plataforma en ambientes de desarrollo y/o certificación para la mejora contínua de la misma, requeriría de un esfuerzo y tiempos importantes, si dicha tarea se lo realizara de manera operativa.

Es así que, con miras a mejorar la experiencia en el aprovisionamiento y consecuente mantenimiento de la plataforma de *iPaaS*, en cualquiera de los ambientes y/o versiones requeridas, se procedió a diseñar una serie de esquemas de todos los componentes de la plataforma, basada en lenguajes y herramientas para la definición de infraestructuras, y que como resultado, entregan un conjunto de arhivos basados en código, para su consumo, ya sea en procesos operativos manuales, o como parte de procesos de automatización

proto-consumer-sink.kamelet.yaml

```yaml
apiVersion: camel.apache.org/v1alpha1
kind: Kamelet
metadata:
  name: proto-consumer-sink
  annotations:
    camel.apache.org/kamelet.icon: "..."
    camel.apache.org/provider: "Apache Software Foundation"
  labels:
    camel.apache.org/kamelet.type: "sink"
    camel.apache.org/requires.runtime: "camel-quarkus"
    camel.apache.org/kamelet.group: "WebHook"
spec:
  definition:
    title: WebHook integration prototype Sink
    description: "Transfer incoming data to a specific webhook"
    required:
      - webhook-api-url
      - webhook-api-key
    properties:
      webhook-api-url:
        title: WebHook Api URL
        description: A secured REST Service URL to post messages
        type: string
      webhook-api-key:
        title: WebHook Api Key
        ...
      request-timeout-ms:
        title: Request Time out
        ...
  dependencies:
    - camel-quarkus:http
  types:
    out:
      mediaType: text/plain
  template:
    from:
      uri: "kamelet:source"
      steps:
        - convert-body-to: "java.lang.String"
        - set-header:
            name: "CamelHttpMethod"
            constant: "POST"
        - set-header:
            name: "Content-Type"
            constant: "application/json"
        - set-header:
            name: "X-Api-Key"
            simple: "{{webhook-api-key}}"
        - remove-headers:
            pattern: "CamelHttp*"
        - do-try:
            steps:
              - to-d: "http:{{webhook-api-url}}?httpClient.SocketTimeout={{request-timeout-ms}}"
            do-catch:
            - exception:
              ...
            - steps:
                - to: "log:error"
```

Figura 3.13: Componente personalizado de *Apache Camel K*

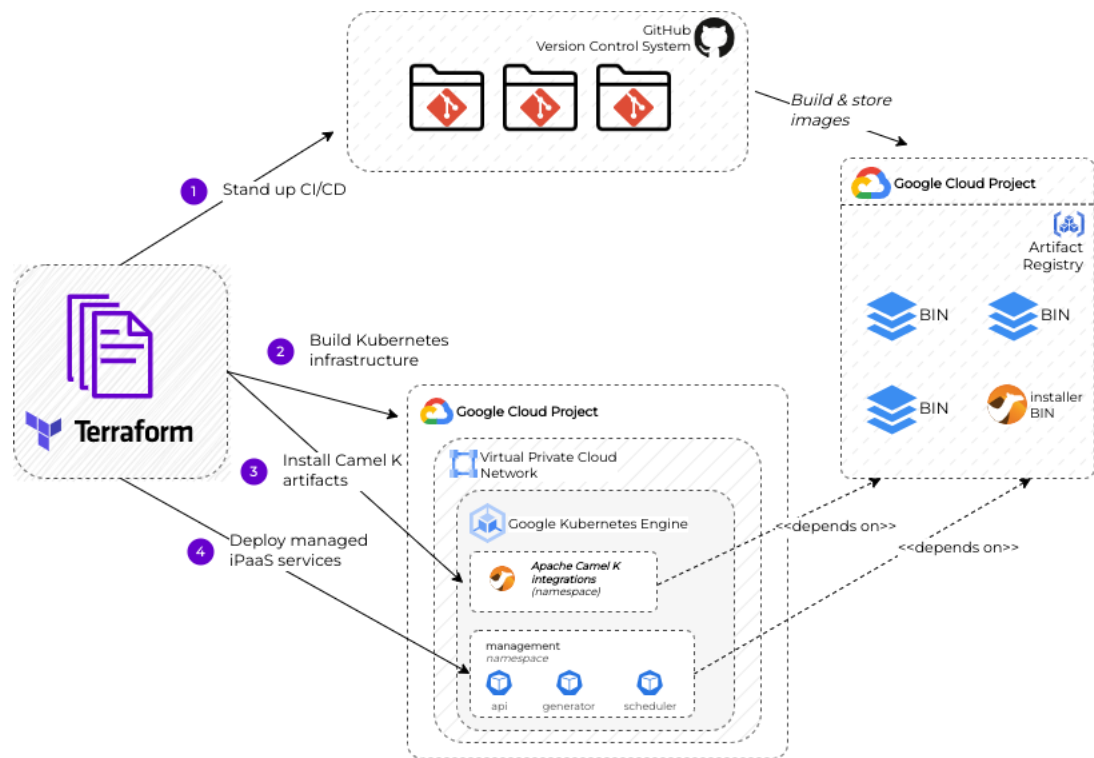


Figura 3.14: Esquema de Infraestructura como Código dispuesta para el *iPaaS*

Tabla 3.2: Agrupamiento y estructura de directorios para recursos Terraform de la plataforma *iPaaS*

Nombre	**Descripción**
gke-artifacts	Directorio que agrupa todos los archivos que definen los recursos de la plataforma
gke-artifacts/ builder-definitions	Subdirectorio con todos los recursos para aprovisionamiento de procesos CI/CD de la plataforma.
gke-artifacts/ kamel-resources	Subdirectorio que mantiene todos los recursos para la instalación y configuración del *Apache Camel K* en plataforma *Kubernetes.*
gke-artifacts/ services	Subdirectorio que mantiene las definiciones de los componentes y servicios desarrollados para complementar el funcionamiento del *iPaaS*
gke-cluster	Directorio que mantiene todos los recursos para la instalación y configuración de la plataforma *Kubernetes* sobre el cual correrá el *iPaaS*

gestionados por *DevOps* a nivel de la entrega y despliegue continuo de los servicios de una Organización.

La herramienta que se eligió para la definición de la infraestructura de la plataforma de *iPaaS* fue *Terraform.* Esta herramienta de tipo *open-source* y elaborada por la empresa *Hashicorp* mantiene un alto nivel de aceptación por parte de los principales proveedores de servicios de cómputo en la nube como *Azure*, *AWS* y *Google Cloud*, por nombrar algunos. De la misma manera, y consecuencia del lenguaje de definición propuesto por el proveedor de la herramienta, llamado *HCL*, junto a una gran cantidad de componentes basados en este lenguaje y disponibles para su utilización en el sitio de registro de *Terraform*, se han desarrollado facilitadores o herramientas de ayuda para mejorar la construcción de los ficheros de infraestructura, en los entornos de desarrollo integrados más populares como *Jetbrains* o *Visual Studio Code.*

Partiendo de las bondades y facilidades existentes a la hora de interactuar con la herramienta previamente seleccionada, se procedió a diseñar la siguiente estructura de archivos, agrupándolas en directorios según la tarea macro a ejecutar durante el aprovisionamiento de la plataforma

3.5.1. Estructura de Directorios

Considerando el proveedor de cómputo en la nube, utilizado por la empresa Location World para mantener su infraestructura y plataforma de negocio, se han elaborado las definiciones tomando en consideración dicho particular. Sin embargo, la elaboración de nuevas definiciones compatibles con cualquier otro proveedor de cómputo en la nube es posible, usando servicios o componentes similares que son utilizados en el aprovisionamiento de la plataforma. En cualquier caso, el grupo de componentes y servicios desarrollados para incorporarlos a la plataforma son agnósticos a cualquier plataforma de cómputo en la nube y podrán ser utilizados sin restructuración alguna.

El proyecto de *Terraform* para la plataforma y según se lo generaliza en la Figura 3.14, está construido por el directorio **gke-artifacts**, que a su vez mantiene dos subdirectorios con los artefactos o recursos necesarios para correr la plataforma *iPaaS*. El directorio **gke-artifacts/builder-definitions** mantiene los recursos que levantarán los flujos para ejecución de entrega y despliegue continuo desde el repositorio en la nube para control de versionamiento de código, como lo es *Github*, y el servicio de *Cloud Build* de la plataforma de *Google Cloud* para la construcción de las imágenes o contenedores a desplegarse en la infraestructura de *Kubernetes*.

El siguiente subdirectorio **gke-artifacts/kamel-resources** contiene los recursos necesarios para la descarga de los componentes de *Apache Camel K*, la configuración del mismo en la infraestructura de *Kubernetes* y el uso del servicio de Registro de Contenedores *Google Container Registry (https://gcr.io)* , y la consecuente instalación del *framework EIP*.

En relación al directorio **gke-cluster**, el mismo mantiene los recursos a nivel del proveedor de cómputo en la nube para el aprovisionamiento de un clúster de *Kubernetes* y su respectivo recurso de red o *Virtual Private Cloud VPC* para la creación de los nodos o máquinas virtuales del clúster como tal.

Por último, se definió el directorio **services**, como el espacio para mantener los recursos de *Kubernetes* que desplegarán los servicios desarrollados como lo son el **API Integration Management**, el **Integration Generator** y el **Integration Scheduler**, dentro de un recurso o *namespace* distinto al que se usará para correr las integraciones de los clientes.

3.5.2. Seguridad de la infraestructura

Partiendo del hecho de que ciertos servicios y componentes mantienen tareas en el aprovisionamiento de ciertos elementos en la infraestructura de *Kubernetes*, como los recursos de tipo *POD* por ejemplo, se estableció el uso

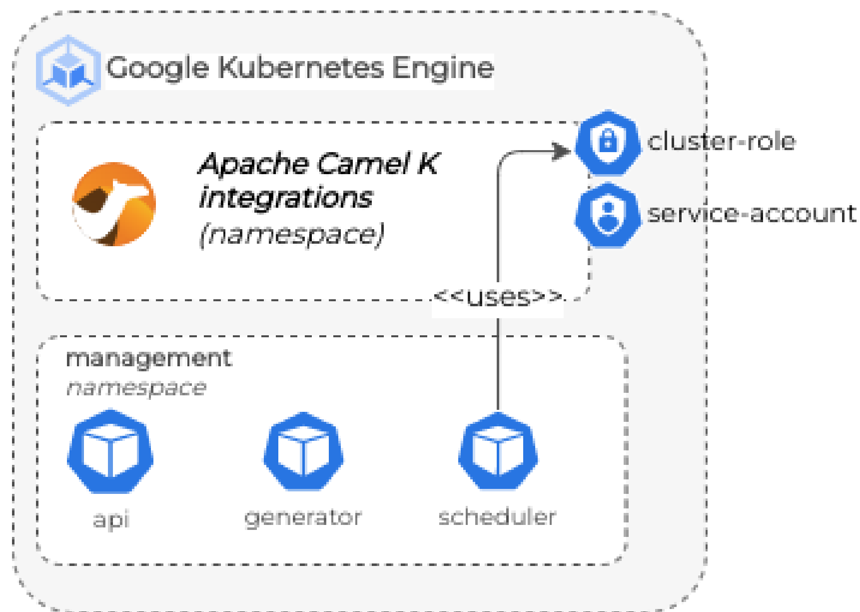


Figura 3.15: Esquema de *namespaces* propuesto en *Kubernetes*

de elementos de seguridad que permitan delimitar el área de acción sobre el cual estos actores ejecutarán dichas tareas de aprovisionamiento.

Considerando las mejores prácticas a la hora de interactuar con infraestructuras como *Kubernetes*, se definió la creación de espacios lógicos o *namespaces*, los cuales pueden apreciarse en la Figura 3.15, para aislar el área de ejecución de los componentes. Por un lado tenemos el *namespace* **management**, en donde se ejecutarán los servicios desarrollados para la autogestión de las integraciones de los clientes, y un *namespace* denominado **integrations**, en donde correrán dichas integraciones de manera aislada al resto de componentes de operación del clúster *Kubernetes*.

Por otro lado, fue necesario la definición de elementos llamados **cuentas de servicios** que se asocian a un grupo muy granular e identificado de permisos o acciones que un componente como el **Integration Scheduler** sería capaz de ejecutar dentro del clúster de *Kubernetes*. Complementario a las cuentas de servicios, se definieron elementos de tipo **roles de clúster** para especificar los permisos exclusivos de ejecución y un objeto de tipo **enlace de rol de clúster** *(ClusterRoleBinding)* para asociar el rol personalizado y la cuenta de servicio atada al elemento *POD* que ejecutará las tareas de aprovisionamiento de integraciones en el *namespace* indicado. En la Figura 3.16, se puede apreciar el esquema *tf* que definen los elementos de seguridad indicados previamente.

scheduler-security-assets.tf

```
resource "kubernetes_cluster_role" "manage_integrations_role" {
  metadata {
    name = "manage_integrations_role"
  }

  rule {
    api_groups=[""]
    verbs = [ "get", "list", "watch", "create", "patch", "update", "delete"
]   resources = [ "pods", "secrets" ]
  }

  rule {
    api_groups = [ "" ]
    verbs =  ["get", "list", "watch"]
    resources = [ "nodes", "namespaces" ]
  }
}

resource "kubernetes_cluster_role_binding" "kamel_role_binding" {
  metadata {
    name = "manage-integrations"
  }
  subject {
    kind = "ServiceAccount"
    name = "ipaas-scheduler-sa"
    namespace = "integrations"
  }

  role_ref {
    api_group = "rbac.authorization.k8s.io"
    kind = "ClusterRole"
    name = "manage_integrations_role"
  }
}
```

Figura 3.16: Ejemplo de esquema terraform que define elementos de seguridad en *Kubernetes*

3.6. Síntesis del capítulo

En este capítulo se ha presentado el diseño para la propuesta de la plataforma de integración con capacidades de autogestión por parte de los clientes que adviertan necesidades de consumo de información; siendo en el caso de Location World, consumo de datos telemáticos de sus vehículos.

Dentro de la propuesta, se han descrito los componentes fundamentales de la plataforma, como por ejemplo, un servicio de tipo *API Rest* para la administración de la metadata de las integraciones a ser aprovisionadas; así como también, los servicios para generación y planificación en la ejecución de los esquemas de integración construidos, dentro de un ambiente para gestión y ejecución de procesos como contenedores, que es *Kubernetes*. Por otro lado, se ha establecido el esquema de datos a usarse con soluciones de

repositorios de datos documentales o *NoSQL* por la flexibilidad que estas soluciones exponen a la hora de mutar las estructuras de datos, según las nuevas funcionalidades o necesidades que requiera la plataforma. Así mismo, se ha propuesto el uso de *frameworks* para el montaje de soluciones de integración empresarial como *Apache Camel K*, para fortalecer el núcleo de la propuesta de *iPaaS* con capacidad de autogestión. Por último, se ha expuesto el diseño del esquema del aprovisionamiento de la infraestructura necesaria y enfocada en el uso de servicios en la nube como *Google Cloud* (el proveedor que utiliza Location World), mediante el uso de herramientas de automatización en el levantamiento de las mismas con herramientas de tipo *IaC* como *Terraform* por ejemplo, junto con propuestas de seguridad necesarias a la hora de aprovisionar y ejecutar los componentes desarrollados para esta propuesta. En el siguiente capítulo, se expondrán las pruebas y evidencias de la ejecución de la *iPaaS*, a través de una prueba de concepto elaborada para el efecto.

Capítulo 4

Resultados y discusión

En este capítulo se establece el contexto y los recursos complementarios necesarios para la elaboración de una prueba de concepto, aplicando las definiciones de diseño y las tecnologías propuestas para la plataforma *iPaaS*, estipuladas en el Capítulo 3.

A continuación, se presentarán los distintos planes de pruebas funcionales aplicados a la *iPaaS*, junto con los resultados obtenidos en cada caso para su posterior discusión.

4.1. Contexto de la prueba de concepto

Con la finalidad de demostrar el éxito que tiene el alcance del diseño de *iPaaS* propuesto y presentado en el Capítulo 3, con capacidades de autogestión por parte de clientes, se procedió con la implementación de una prueba de concepto para tal fin.

Para esta prueba de concepto, se abarcó el desarrollo de los componentes personalizados propuestos en el diseño como son el *Integration Generator*, el *Integration Scheduler* y de los *API Admin* y *API Integration Management*, estos dos últimos encapsulados en un mismo servicio para optimizar del levantamiento de la prueba en sí.

Adicionalmente, se definió el uso de servicios y/o soluciones de cómputo en la nube, a través del proveedor de *Google Cloud*, para los componentes de tipo Repositorio *NoSQL*, el sistema de cola de mensajes para manejo de la comunicación inter-proceso, el gestor de almacenamiento de los archivos que definen las integraciones y por último, la plataforma de orquestación de contenedores de *Kubernetes*. Se justifica el uso del proveedor debido a que la infraestructura tecnológica de la empresa *Location World* se hospeda allí y es a quien va dirigida la solución de esta integración. Sin embargo, no se descarta la posibilidad de que esta propuesta de *iPaaS* pueda ser montada en cualquier proveedor de nube que se requiera.

Por último, se desarrollaron recursos auxiliares para la prueba de concepto (y fuera del alcance de la propuesta de *iPaas*) que simulen la fuente de datos de donde se alimentará la integración definida por un cliente y el servicio destino de dicho cliente, que recibirá la información en cuestión. Como detalle técnico, se estableció como la fuente primaria de datos, el sistema de mensajería basado en colas de *RabbitMQ*, y un servicio de tipo *API Rest* con una simple capa de autenticación y autorización basada en un *API-Key*.

4.2. Pruebas funcionales de la plataforma

La ejecución de las pruebas funcionales sobre la plataforma *iPaaS* desplegada bajo la prueba de concepto, se centró en 3 aspectos principales:

i) El levantamiento de la infraestructura propuesta con la herramienta *IaC* de *Terraform*

ii) El proceso de autogestión de una integración, desde su registro a través del servicio de *API* de administración, como su posterior despliegue en la plataforma de *Kubernetes*.

iii) El procesamiento de un mensaje por parte de la plataforma, usando los recursos auxiliares de la cola de mensajes *RabbitMQ* y el servicio *webhook* de tipo *API Rest* junto a un repositorio emulado de tipo *NoSQL*.

4.2.1. Prueba funcional del despliegue de plataforma

Descripción

Se inicia con la ejecución de los comandos de *Terraform* dentro de cada directorio definido para el esquema *IaC* descrito en el Capítulo 3, Tabla 3.2.

Resultado esperado

1. El clúster de *Kubernetes* se despliega en plataforma de *Google Cloud*
2. Los componentes de *Apache Camel K* se instalan y ejecutan en clúster *Kubernetes*
3. Los componentes personalizados para el *iPaaS* se instalan y ejecutan en clúster *Kubernetes*

Código de ejecución de la prueba

Listing 4.1: Sentencia *Terraform* para creación de clúster *Kubernetes*

```
/> cd gke-cluster
gke-cluster/> ls -la
-rw-r--r-- 1 some_user some_user 1.4K Apr 5 20:50 gke.tf
-rw-r--r-- 1 some_user some_user 224 Mar 8 21:15 main.tf
-rw-r--r-- 1 some_user some_user 503 Mar 9 17:16 outputs.tf
-rw-r--r-- 1 some_user some_user 181 Apr 6 12:19 terraform.tfstate
-rw-r--r-- 1 some_user some_user 23K Apr 6 12:11 terraform.tfstate.backup
-rw-r--r-- 1 some_user some_user 289 Mar 31 15:27 terraform.tfvars
-rw-r--r-- 1 some_user some_user 696 Mar 9 14:58 variables.tf
-rw-r--r-- 1 some_user some_user 356 Mar 9 11:37 vpc.tf
gke-cluster/> terraform init
gke-cluster/> terraform plan
gke-cluster/> terraform apply -auto-approve
```

Listing 4.2: Sentencia *Terraform* para instalación de *Apache Camel K*

```
/> cd gke-artifacts/kamel-resources
gke-artifacts/kamel-resources/> ls -la
-rw-r--r-- 1 some_user some_user 2.6K Mar 30 15:35 deployment.tf
-rw-r--r-- 1 some_user some_user 1.1K Mar 30 14:54 main.tf
-rw-r--r-- 1 some_user some_user 182 Apr 6 12:19 terraform.tfstate
-rw-r--r-- 1 some_user some_user 36K Apr 6 12:01 terraform.tfstate.backup
-rw-r--r-- 1 some_user some_user 221 Mar 30 15:34 terraform.tfvars
-rw-r--r-- 1 some_user some_user 492 Mar 30 15:34 variables.tf
gke-artifacts/kamel-resources/> terraform init
gke-artifacts/kamel-resources/> terraform plan
gke-artifacts/kamel-resources/> terraform apply -auto-approve
```

Listing 4.3: Sentencia *Terraform* para instalación de *Integration API Management*

```
/> cd gke-artifacts/services/ipaas-management
gke-artifacts/services/ipaas-management/> ls -la
-rw-r--r-- 1 some_user some_user 932 Apr 5 22:05 deployment.tf
-rw-r--r-- 1 some_user some_user 1.1K Mar 30 17:09 main.tf
-rw-r--r-- 1 some_user some_user 181 Apr 6 12:19 terraform.tfstate
-rw-r--r-- 1 some_user some_user 23K Apr 6 11:57 terraform.tfstate.backup
-rw-r--r-- 1 some_user some_user 261 Mar 30 17:11 variables.tf
gke-artifacts/services/ipaas-management/> terraform init
gke-artifacts/services/ipaas-management/> terraform plan
gke-artifacts/services/ipaas-management/> terraform apply -auto-approve
```

Listing 4.4: Sentencia *Terraform* para instalación de *Integration Generator*

```
/> cd gke-artifacts/services/ipaas-generator
gke-artifacts/services/ipaas-generator/> ls -la
-rw-r--r-- 1 some_user some_user 932 Apr 5 22:05 deployment.tf
-rw-r--r-- 1 some_user some_user 1.1K Mar 30 17:09 main.tf
-rw-r--r-- 1 some_user some_user 181 Apr 6 12:19 terraform.tfstate
-rw-r--r-- 1 some_user some_user 23K Apr 6 11:57 terraform.tfstate.backup
-rw-r--r-- 1 some_user some_user 261 Mar 30 17:11 variables.tf
gke-artifacts/services/ipaas-generator/> terraform init
gke-artifacts/services/ipaas-generator/> terraform plan
gke-artifacts/services/ipaas-generator/> terraform apply -auto-approve
```

Listing 4.5: Sentencia *Terraform* para instalación de *Integration Scheduler*

```
/> cd gke-artifacts/services/ipaas-scheduler
gke-artifacts/services/ipaas-scheduler/> ls -la
-rw-r--r-- 1 some_user some_user 932 Apr 5 22:05 deployment.tf
-rw-r--r-- 1 some_user some_user 1.1K Mar 30 17:09 main.tf
-rw-r--r-- 1 some_user some_user 181 Apr 6 12:19 terraform.tfstate
-rw-r--r-- 1 some_user some_user 23K Apr 6 11:57 terraform.tfstate.backup
-rw-r--r-- 1 some_user some_user 261 Mar 30 17:11 variables.tf
gke-artifacts/services/ipaas-scheduler/> terraform init
gke-artifacts/services/ipaas-scheduler/> terraform plan
gke-artifacts/services/ipaas-scheduler/> terraform apply -auto-approve
```

Resultado obtenido

Kubernetes clusters CREATE DEPLOY REFRESH

OVERVIEW OBSERVABILITY COST OPTIMIZATION

Filter Enter property name or value

Status	Name	Location	Number of nodes	Total vCPUs	Total memory
✓	ipaas-gke	us-central1-a	2	4	15 GB

(a) Clúster Kubernetes corriendo

Workloads REFRESH DEPLOY DELETE

Cluster: ipaas-gke Namespace: integrations RESET SAVE

Workloads are deployable units of computing that can be created and managed in a cluster.

OVERVIEW OBSERVABILITY COST OPTIMIZATION

Filter Is system object : False camel* Filter workloads

Name	Status	Type	Pods	Namespace
camel-k-kit-cgn31rp1o817lgv7ndl0-builder	Succeeded	Pod	0/1	integrations
camel-k-operator	OK	Deployment	1/1	integrations

(b) Apache Camel K corriendo

Workloads are deployable units of computing that can be created and managed in a cluster.

OVERVIEW OBSERVABILITY COST OPTIMIZATION

Filter Is system object : False Filter workloads

Name	Status	Type	Pods	Cluster
camel-k-kit-cgn31rp1o817lgv7ndl0-builder	Succeeded	Pod	0/1	ipaas-gke
camel-k-operator	OK	Deployment	1/1	ipaas-gke
ipaas-generator	OK	Deployment	1/1	ipaas-gke
ipaas-management	OK	Deployment	1/1	ipaas-gke
ipaas-scheduler	OK	Deployment	1/1	ipaas-gke

(c) Componentes personalizados corriendo

Figura 4.1: Componentes y recursos *iPaaS* corriendo en *Google Kubernetes Engine*

4.2.2. Prueba funcional de autogestión de integración

Descripción

Se inicia con la creación de las entidades *Datasource* e *Integration*, así como el enlace del cliente con el *Datasource* para ser permitido de usarlo en la integración.

Resultado esperado

1. La entidad *Datasource* se registra exitosamente en el repositorio principal.
2. La entidad *Integration* se registra exitosamente en el repositorio principal.
3. El enlace entre el cliente y el *Datasource* se registra exitosamente en el repositorio principal.
4. Los archivos de las definiciones de la integración son creadas y cargadas exitosamente por el servicio *Integration Generator* en el almacenamiento principal de la plataforma.
5. La lectura de los archivos y su carga en ambiente *Kubernetes* es ejecutada exitosamente por el servicio *Integration Scheduler*.

Código de ejecución de la prueba

Listing 4.6: Sentencia *CURL* para creación de entidad *Datasource*

```
curl -X POST \
  'http://localhost:9090/admin/datasources' \
  --header 'Accept: */*' \
  --header 'Content-Type: application/json' \
  --data-raw '{
  "type": "RabbitMQ",
  "name": "Mi first datasource",
  "addresses": [
    "RabbitMQ-sv.default:5672"
  ],
  "username": "some_user",
  "password": "xxxxxxxx",
  "exchangeName": "gps_data"
}'
```

Listing 4.7: Sentencia *CURL* para enlazar cliente con *Datasource*

```
    curl -X POST \
  'http://localhost:9090/admin/datasources/ZvNXYpfvEnHN2MhsXvqM/clients' \
```

```
  --header 'Accept: */*' \
  --header 'Content-Type: application/json' \
  --data-raw '["xyz1234pkil"]'
```

Listing 4.8: Sentencia *CURL* para creación de entidad *Integration*

```
    curl -X POST \
  'http://localhost:9090/integrations' \
  --header 'Accept: */*' \
  --header 'Content-Type: application/json' \
  --data-raw '{
  "name":"SOme integration name",
  "dataSourceId":"ZvNXYpfvEnHN2MhsXvqM",
  "destiny":{
    "type":"WebHook",
    "url":"webhook-sv.default:9090/vehicles/",
    "security":{
      "type":"ApiKey",
      "apiKey":"a2025c2fe552f04aa2df..."
    }
  }
}'
```

Resultado obtenido

(a) Datasource

(b) Integration

Figura 4.2: Entidades *Datasource* e *Integration* creadas en repositorio principal

```
❯ kubectl logs -f deploy/ipaas-generator -n integrations
INFO:__main__:apiVersion: v1
kind: Secret
metadata:
  name: 1-it-secret-umhzxwn9jmli95cyftuy
stringData:
  1-it-secret-umhzxwn9jmli95cyftuy.properties: |-
    camel.kamelet.rabbitmqv2-source.addresses=rabbitmq-sv.default:5672
    camel.kamelet.rabbitmqv2-source.queue=q-xyz1234pkil.umHZxwN9jMli95cyFTuY
    camel.kamelet.rabbitmqv2-source.username=some_user
    camel.kamelet.rabbitmqv2-source.password=no.seas.asi.123
    camel.kamelet.rabbitmqv2-source.exchangeType=fanout
    camel.kamelet.rabbitmqv2-source.exchangeName=gps_data
    camel.kamelet.rabbitmqv2-source.durable=false
    camel.kamelet.rabbitmqv2-source.autoDetele=false

    camel.kamelet.proto-consumer-sink.webhook-api-url=webhook-sv.default:9090/vehicles/
    camel.kamelet.proto-consumer-sink.webhook-api-key=a2025c2fe552f04aa2dfac0d7b2c3dda5099977894d04c1ce14b659e771c3747
type: Opaque

INFO:__main__:-----------------------------------------------------
INFO:__main__:apiVersion: camel.apache.org/v1alpha1
kind: KameletBinding
metadata:
  name: 2-it-definition-umhzxwn9jmli95cyftuy
spec:
  integration:
    configuration:
    - type: secret
      value: 1-it-secret-umhzxwn9jmli95cyftuy
  sink:
    ref:
      apiVersion: camel.apache.org/v1alpha1
      kind: Kamelet
      name: proto-consumer-sink
  source:
    ref:
      apiVersion: camel.apache.org/v1alpha1
```

(a) Ejecución *Integration Generator*

Bucket details

ipaas_integration_assets

Location	Storage class	Public access	Protection
us-central1 (Iowa)	Standard	Subject to object ACLs	None

OBJECTS CONFIGURATION PERMISSIONS PROTECTION LIFECYCLE OBSERVABILITY NEW

Buckets > ipaas_integration_assets > umHZxwN9jMli95cyFTuY

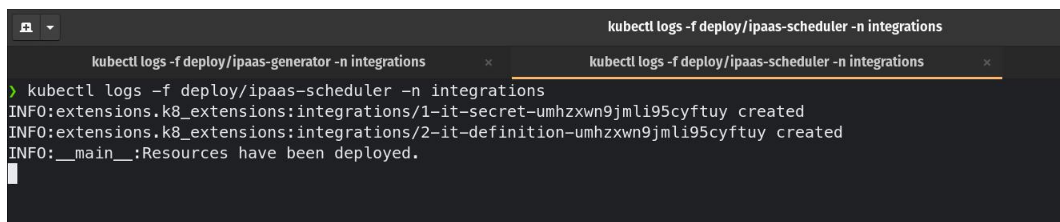
UPLOAD FILES UPLOAD FOLDER CREATE FOLDER TRANSFER DATA MANAGE HOLDS DOWNLOAD DELETE

Filter by name prefix only Filter Filter objects and folders

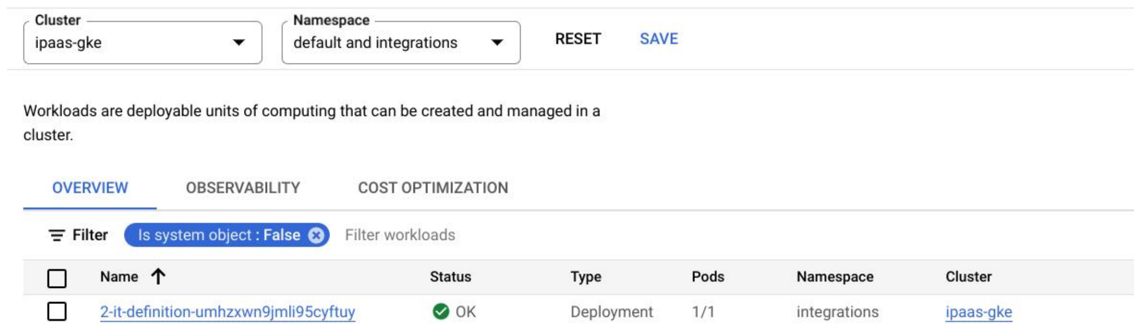
Name	Size	Type	Created	Storage class	Last modified	Public access
1-it-secret-umhzxwn9jmli95cyftuy...	843 B	application/yaml	Apr 6, 2023, 9:43:35 AM	Standard	Apr 6, 2023, 9:43:35 AM	Not public
2-it-definition-umhzxwn9jmli95cyf...	445 B	application/yaml	Apr 6, 2023, 9:43:35 AM	Standard	Apr 6, 2023, 9:43:35 AM	Not public

(b) Almacenamiento definiciones en *Google Storage*

Figura 4.3: Definiciones de integración creadas y cargadas a almacenamiento principal



(a) Ejecución servicio *Integration Scheduler*



(b) Ejecución integración en *Kubernetes*

Figura 4.4: Procesamiento de archivos de integración y carga en *Kubernetes*

4.2.3. Pruebas funcional de funcionamiento del *iPaaS*

Descripción

Se inicia con el envío de un mensaje hacia la integración previamente creada y ejecutándose en la plataforma *iPaaS*, usando la misma interfaz de administración de *RabbitMQ* para el efecto de la prueba.

Resultado esperado

1. El mensaje es recibido desde la fuente de datos y procesada hacia el destino
2. El destino recibe el mensaje y lo almacena en su repositorio persistente

Código de ejecución de la prueba

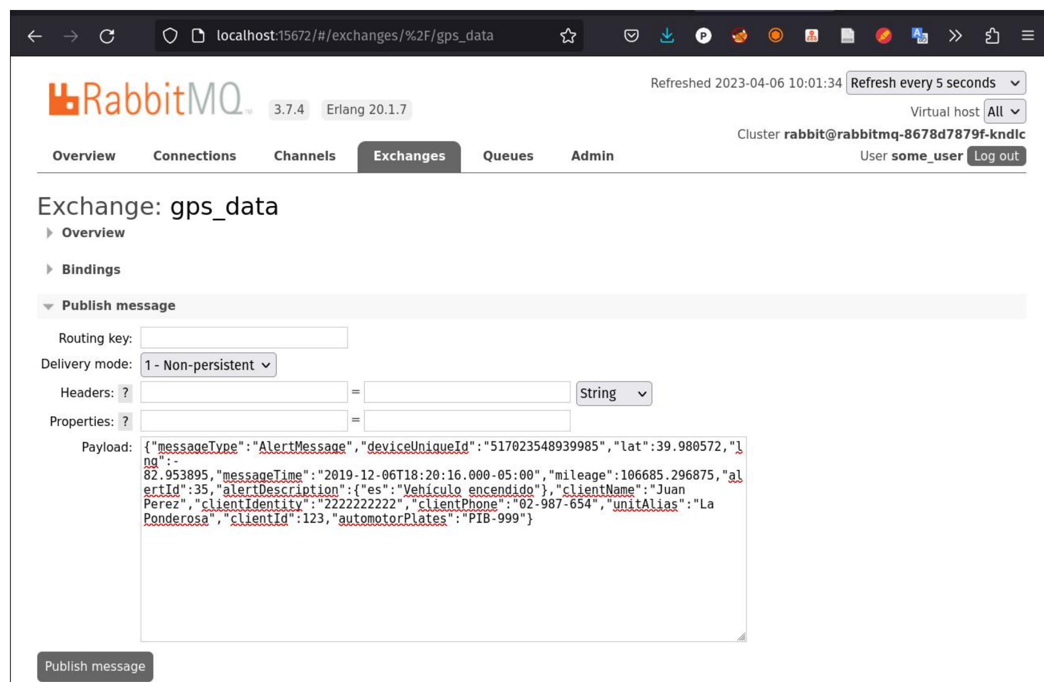


Figura 4.5: Envío de mensaje a través del sistema de colas

(a) Ejecución integración

(b) Recepción del *Webhook* del cliente

(c) Mensaje almacenado en el lado del cliente

Figura 4.6: Procesamiento y recepción de mensaje por *iPaaS* y destino

Resultado obtenido

4.3. Análisis resultados y discusión

A partir de las evidencias obtenidas durante la ejecución de las pruebas funcionales, sobre la prueba de concepto implementada a partir del diseño propuesto para la plataforma *iPaaS*, se analizaron los siguientes aspectos:

i) Se llegó a cumplir con el criterio de autogestión de los procesos de integración por parte de clientes, gracias a la definición de tipo *DSL*

Figura 4.7: Integración continua de las integraciones con *Google Cloud Build* y *Github*

soportada por *Apache Camel*, y de la capacidad de despliegue dinámico entregado por la plataforma de *Kubernetes*.

ii) Se llegó a determinar que gracias al manejo de *Apache Camel* como la base en la construcción de procesos de integración, es posible la extensibilidad de nuevas funcionalidades en las integraciones, sin que se requiera un cambio de fondo en todo el diseño propuesto para la plataforma de *iPaaS*, como por ejemplo, inclusión de reglas de filtrado, transformación y agregación de los datos previo a su entrega en el destino.

iii) Se presentó la necesidad de levantar procesos de entrega continua de los componentes *Integration Generator*, *Integration Scheduler*, *Integration API Management*, para completar todo el proceso de despliegue a través de *Terraform* (véase la Figura 4.7). Este punto es muy importante, sobre todo a la hora de obtener el control sobre las mejoras y/o mantenimiento requerido de la plataforma *iPaaS* en un futuro.

iv) Si bien el criterio de seguridad es muy importante a la hora de diseñar e implementar cualquier sistema o plataforma, su aplicabilidad es una tarea que demanda tiempo y esfuerzos a corto, mediano y largo plazo. Para el caso de la prueba de concepto, criterios como la encripción de datos y el manejo granular de mecanismo de control de accesos, a nivel del proceso para levantamiento de la infraestructura con *Terraform*, no fueron aplicados en su totalidad a la hora de levantar la prueba de concepto. Sin embargo, se consideran importante respetar estos criterios a la hora de montar la plataforma *iPaaS* dentro entornos productivos y funcionales.

v) Considerando que la propuesta de *iPaaS* incluía el uso de algunos servicios de tipo *open source* o de uso gratuito como respositorios NoSQL Mongo, gestores de almacenamiento como *Minio* o el uso de sistema de mensajería por colas como *RabbtitMQ*, fue evidente los beneficios otorgados

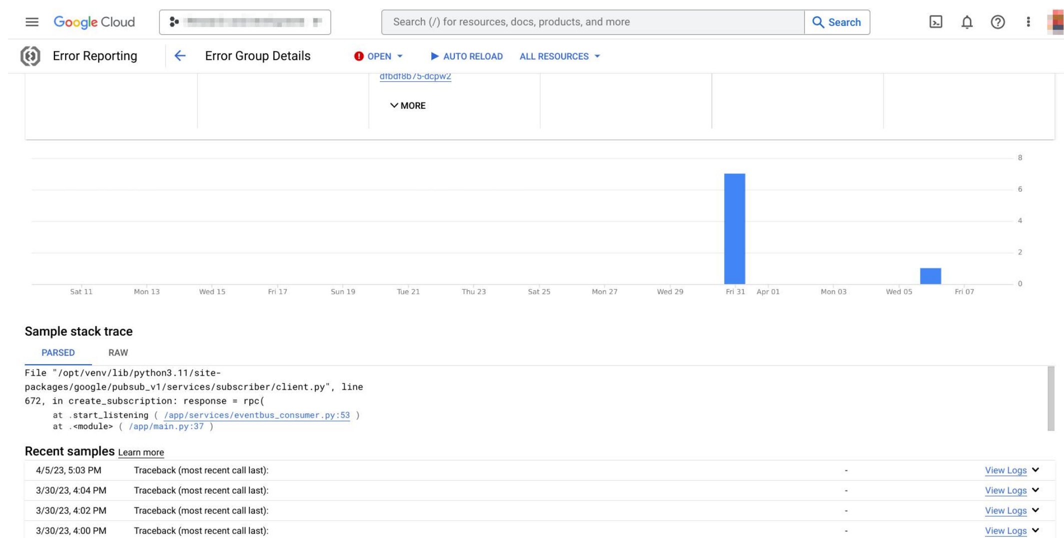


Figura 4.8: Control de errores mediante servicios de la nube

por soluciones en la nube, y en éste caso con *Google Cloud*, como los servicios de *PubSub*, *Firestore* y *Cloud Storage*; todos ellos a la hora de optimizar tiempos y abstraer la complejidad de administración presentes en una autogestión de las mismas.

4.3.1. Control de errores del *iPaaS*

Uno de los puntos que se destacó durante las pruebas funcionales ejecutadas sobre la plataforma fue el control y monitoreo del comportamiento y los errores producidos durante su ejecución. Bajo el diseño planteado para la plataforma de integración, existen distintos puntos en donde el flujo principal de ejecución puede fallar, y en esos casos, la detección oportuna y el tratamiento adecuado a estas incidencias es fundamental.

Se llegó a aprovechar toda la instrumentalización propia en el monitoreo y control de errores que el proveedor en la nube de *Google Cloud* dispone para todos sus servicios de manera automática y centralizada. Cada uno de los logs de errores generados por los distintos componentes, dentro de la plataforma de *Google Kubernetes Engine*, llegaron a ser registrados dentro del grupos de servicios destinados para el efecto: *Cloud Monitoring* y *Error Reporting*. De estos dos servicios, se determinó que con el modelo o patrón de datos pertinente, es posible la generación de alertas que permitan informar de situaciones críticas que requieran de una inmediata atención dentro del *iPaaS*. En la siguiente Figura 4.8, se puede apreciar el control automático de errores para la plataforma, con el uso de soluciones en la nube:

4.4. Síntesis del capítulo

Dentro de este capítulo, se llegó a demostrar la efectividad de la solución de *iPaaS* planteada en el Capítulo 3 para la autogestión de integraciones por parte de los clientes.

Así mismo, se evidenció las ventajas inherentes al incorporar marcos de trabajo enfocados a *EIP*, como lo es en este caso *Apache Camel*, gracias a la extensibilidad proporcionada para las integraciones, con el uso de definiciones estandarizadas y listas para su uso.

De igual manera, se evidenció los beneficios adquiridos por la incorporación de servicios y soluciones de infraestructura en la nube, tanto para la comunicación inter-procesos, el almacenamiento centralizado de recursos, los repositorios de datos y en especial, el control y monitoreo de errores, permitiendo un solución más robusta para el ámbito empresarial.

En el siguiente capítulo, se elaborarán las conclusiones pertinentes al desarrollo de este proyecto de tesis y de los resultados y análisis elaborados del mismo.

Capítulo 5

Conclusiones

En este último Capítulo, se formulan tanto las conclusiones generales a partir de las evidencias y/o resultados obtenidos y analizados en el Capítulo 4 de este trabajo,, así como una serie de recomendaciones que definirán el ciclo de vida de la propuesta de *iPaaS* a corto, mediano y largo plazo.

5.1. Conclusiones

Luego de la presentación sobre los resultados obtenidos en el Capítulo 4 de este trabajo, se destaca el importante nivel de contribución otorgado por la propuesta de *iPaaS* presentada, y complementada con la capacidad de autogestión, para que las empresas proveedoras de servicios en general, puedan incorporarlos en sus plataformas y disponibilizar de un canal más de servicio, con el que sus clientes, puedan interactuar a la hora de incorporar procesos de integración contra sus sistemas y/o plataformas tradicionales, en el tratamiento de la información.

Por otra parte, la incorporación de marcos de trabajo de tipo *EIP* como es el caso de *Apache Camel*, suponen un gran refuerzo en los cimientos que darán soporte a la plataforma de integración esperada, proporcionando una interfaz madura y estándar en la definición, diseño y desarrollo de los componentes que fungirán con la tarea de recolectar, validar, transformar y entregar, los distintos tipos de datos que pueda generarse dentro de la plataforma de un proveedor de servicios.

Así mismo, y con base en los resultados obtenidos y presentados en las pruebas funcionales ejecutadas sobre la prueba de concepto, se validó el funcionamiento exitoso con la propuesta de la plataforma *iPaaS*, desde su despliegue en un ambiente orquestado y clusterizado como *Kubernetes*, pasando por la autogestión de una integración mediante una interfaz de tipo *API*, hasta llegar a su ejecución en la entrega de datos desde una fuente como *RabbitMQ* hasta el sistema del cliente (representado por un *WebHook* y repositorio *NoSQL*).

Por último, y considerando como evidencia los pasos descritos en la prueba funcional del despliegue de plataforma, en el capítulo 4, se comprueba el éxito obtenido con el desarrollo de la definición para el proceso de automatización de la plataforma de *iPaaS*, partiendo del uso de herramientas de tipo *IaC*, como lo es *Terraform*. Así mismo, se corroboró la gran capacidad de estas herramientas, a la hora de levantar cualquier tipo de plataforma y de su infraestructura subyacente en cuestión de minutos, y bajo los distintos criterios o necesidades en términos de ambientes que requieran de su disponibilidad, dentro de una empresa.

El desarrollo de este trabajo ha permitido visibilizar una propuesta en el manejo de plataformas de integración autogestionadas con el propósito de optimizar los costos inherentes dentro del desarrollo a medida de procesos de integración, tanto en términos de recurso humano (desarrolladores), como del tiempo requerido para su ejecución. Con la presencia de soluciones de infraestructura, de servicios y/o cómputo en la nube, disponibles a través de algunos proveedores de gran renombre como *Amazon*, *Microsoft* o *Google*, y la madurez adquirida de ciertas tecnologías en la orquestación de arquitecturas basada en microservicios como lo son *Docker* y *Kubernetes*, se hace evidente la eficiencia con la que platformas de tipo *iPaaS* pueden ser aprovechadas al

máximo dentro de las operaciones de las organizaciones, y la eficacia con la que éstas pueden operar para garantizar un adecuado nivel de servicio.

5.2. Recomendaciones

Luego de presentadas las conclusiones formuladas sobre el desarrollo de este trabajo, se exponen las siguientes recomendaciones con el afán de conseguir la madurez sobre esta propuesta y que amplie los beneficios para quienes las implementen en sus infraestructuras.

Respecto al tema de costos, se considera importante enfocar el manejo de la infraestructura necesaria de la *iPaaS* con servicios de cómputo en la nube. La justificación se basa en el hecho de que en la nube, la gobernanza de los costos está estrechamente relacionado a la cantidad de recursos utilizados en un espacio de tiempo determinado. Este concepto de pago bajo demanda permite una mejor optimización de los valores monetarios que deben ser invertidos para mantener una infraestructura, mitigando los escenarios de sub-utilización y sobre-utilización de los mismos, que en muchos de los casos, se presentan con mayor frecuencia en infraestructuras locales. Se complementa a esta recomendación, el manejo de servicios en la nube tales como repositorios, sistemas de mensajería y gestores de objetos (para el manejo de archivos), basados en el hecho de que los costos de utilización son sumamente flexibles (incluso llegando a pagar casi nada por el concepto de capa gratuita que cada uno maneja) y que abstrae de complejidad y tiempo en su administración.

En relación al criterio de seguridad de la plataforma, la propuesta de *iPaaS*, incluyó en su diseño aspectos de seguridad sobre el manejo de los datos, tanto a nivel de almacenamiento, como en el entorno de despliegue donde se interactúa con la misma, mediante el uso de mecanismos y elementos de encripción como por ejemplo, el uso de objetos de tipo *Secret* dentro de la plataforma de orquestación de *Kubernetes*. No obstante y a pesar de que la prueba de concepto elaborada en el Capítulo 4, omitió algunos de ellos por asuntos netamente demostrativos, es importante cumplir y alinear los criterios propuestos con aquellos establecidos por las empresas y organizaciones bajo las normas de seguridad de la información que hayan sido incorporadas en sus procesos. Se incluye a esta recomendación, la evaluación del uso de soluciones de tipo *SaaS*, como el *Google Key Management Service* por ejemplo, que cumplen rigurosamente con estándares que garanticen la correcta aplicación de mecanismos seguros de cifrado. Por otro lado, la evaluación e incorporación de servicios de administración del acceso de identidades, como una capa de control adicional sobre quién o qué puede ejecutar una determinada acción dentro de este tipo de plataformas, es muy recomendable como un aspecto más de seguridad.

Finalmente, se considera de gran importancia el seguir invirtiendo en

tiempo y esfuerzo humano necesario, para ampliar las funcionalidades y afinar algunas de las ya existentes en la propuesta de *iPaaS* entregada que agreguen mayor valor a las empresas de cara a nuevos servicios y/o tendencias tecnológicas que demanden estar a la vanguardia para con los consumidores de las mismas.

Bibliografía

C. D. Abravanel. What is connected car data? [Online]. https://otonomo.io/blog/connected-car-data/, Jun 2021.

T. Apache-Software-Foundation. Architecture. [Online]. https://camel.apache.org/camel-k/1.10.x/architecture/architecture.html.

R. Balasubramanian, A. Libarikian, and D. McElhaney. Insurance 2030—the impact of ai on the future of insurance. page 10, Mar 2021.

B. Campbell. *The Definitive guide to AWS infrastructure automation: craft infrastructure-as-code solutions*. Apress, Berkeley, CA, 2020. ISBN 978-1-4842-5397-7.

J. Campbell. Comparación entre kubernetes y docker. [Online]. https://www.atlassian.com/es/microservices/microservices-architecture/kubernetes-vs-docker.

T. Cerny, M. J. Donahoo, and M. Trnka. Contextual understanding of microservice architecture: current and future directions. *ACM SIGAPP Applied Computing Review*, 17(4):29–45, Jan 2018. ISSN 1559-6915, 1931-0161. doi: 10.1145/3183628.3183631.

G. Cloud. Kubernetes: Google kubernetes engine (gke) | google kubernetes engine (gke). [Online].https://cloud.google.com/kubernetes-engine?hl=es-419, a.

G. Cloud. ¿qué es pub/sub? | documentación de cloud pub/sub | google cloud. [Online] https://cloud.google.com/pubsub/docs/overview?hl=es-419, b. Accessed: 2023-02-09.

E. Commission. What does the general data protection regulation (gdpr) govern? [Online].https://ec.europa.eu/info/law/law-topic/data-protection/reform/what-does-general-data-protection-regulation-gdpr-govern_en. Accessed: 2022-09-28.

J. D'Emic, V. Romero, and D. Dossot. *Mule in action*. Simon and Schuster, 2014.

D. L. Freire, R. Z. Frantz, F. Roos-Frantz, and S. Sawicki. Survey on the run-time systems of enterprise application integration platforms focusing on performance. *Software: Practice and Experience*, 49(3):341–360, Mar 2019. ISSN 0038-0644, 1097-024X. doi: 10.1002/spe.2670.

M. Howard. Terraform – automating infrastructure as a service. (arXiv:2205.10676), May 2022. doi: https://doi.org/10.48550/arXiv.2205.10676. URL http://arxiv.org/abs/2205.10676. arXiv:2205.10676 [cs].

C. IBM. *Cost of a Data Breach - Report 2022*. United States, Jul 2022. URL https://www.ibm.com/downloads/cas/3R8N1DZJ.

J. Lewis and M. Fowler. Microservices: a definition of this new architectural term. *MartinFowler. com*, 25:14–26, 2014.

luisclausin@gmail.com. ¿qué es mulesoft? ¿componentes de anypoint platform?, Mar 2020. URL https://www.nts-solutions.com/blog/salesforce-mulesoft-que-es.html.

M. Marian. ipaas: Different ways of thinking. *Procedia Economics and Finance*, 3:1093–1098, 2012. ISSN 22125671. doi: 10.1016/S2212-5671(12)00279-1.

Microsoft. Servicio de kubernetes administrado (aks) | microsoft azure. [Online].https://azure.microsoft.com/es-es/products/kubernetes-service/.

K. Morris. *Infrastructure as code*. O'Reilly Media, 2020.

I. A. Nebel. *Arquitectura de Microservicios para Plataformas de Integración*. Tesis de maestría, Universidad de la República (Uruguay). Facultad de Ingeniería, Uruguay, 2019.

T. Neifer, D. Lawo, P. Bossauer, and A. Gadatsch. Decoding ipaas: Investigation of user requirements for integration platforms as a service:. In *Proceedings of the 18th International Conference on e-Business*, page 47–55, Online Streaming, — Select a Country —, 2021. SCITEPRESS - Science and Technology Publications. ISBN 978-989-758-527-2. doi: 10.5220/0010626000470055. URL https://www.scitepress.org/DigitalLibrary/Link.aspx?doi=10.5220/0010626000470055.

J.-B. Onofré. *Mastering Apache Camel*. Packt Publishing Ltd, 2015.

S. Palanimalai and I. Paramasivam. An enterprise oriented view on the cloud integration approaches – hybrid cloud and big data. *Procedia Computer Science*, 50:163–168, Jan 2015. ISSN 1877-0509. doi: 10.1016/j.procs.2015.04.079.

N. Serrano, J. Hernantes, and G. Gallardo. Service-oriented architecture and legacy systems. *IEEE Software*, 31(5):15–19, Sep 2014. ISSN 1937-4194. doi: 10.1109/MS.2014.125.

A. W. Services. Kubernetes en aws | amazon web services. [Online].https://aws.amazon.com/es/kubernetes/.

J. Shah and D. Dubaria. Building modern clouds: Using docker, kubernetes & google cloud platform. In *2019 IEEE 9th Annual Computing and Communication Workshop and Conference (CCWC)*, pages 0184–0189, 2019. doi: 10.1109/CCWC.2019.8666479.