UNIVERSIDAD DE CHILE
FACULTAD DE CIENCIAS FÍSICAS Y MATEMÁTICAS
DEPARTAMENTO DE CIENCIAS DE LA COMPUTACIÓN

CHASKI 2.0: FINALIZACIÓN DESARROLLO DE UN REPARTIDOR DE MENSAJES PARA ADERESO

MEMORIA PARA OPTAR AL TÍTULO DE
INGENIERO CIVIL EN COMPUTACIÓN

ALEX JAVIER ÁLVAREZ TOLEDO

PROFESOR GUÍA:
JOSÉ PINO URTUBIA

MIEMBROS DE LA COMISIÓN:
ALEJANDRO HEVIA ANGULO
CESAR GUERRERO SALDIVIA

SANTIAGO DE CHILE
2022

CHASKI 2.0: FINALIZACIÓN DESARROLLO DE UN REPARTIDOR DE MENSAJES PARA ADERESO

La presente tesis contiene el proceso de desarrollo del servicio denominado Chaski, que consiste en un repartidor de mensajes que resuelve, esencialmente, el problema productor consumidor. Adereso, la empresa que motivó el desarrollo del proyecto, necesita usar el servicio en su proceso productivo, requiere mejoras de Chaski sustanciales en lo que respecta a: robustez del servicio, monitoreo del servicio y persistencia de los datos.

Chaski 1.0, el prototipo desarrollado por el autor de la tesis en su práctica profesional II, es un servicio serverless repartidor de mensajes, que se compone de: una interfaz de entrada HTTP donde los producciones de mensajes puede entregar los datos; un buffer de datos donde se almacenan los mensajes; y funciones lógicas que conectan estos componentes de la solución y generan la consulta HTTP para hacer llegar el mensaje a su destino, el consumidor. Todo esto se logra haciendo uso de servicios de Amazon Web Services, lo que asegura la escalabilidad del servicio.

Chaski 2.0, el proyecto desarrollado en esta memoria, busca complementar el prototipo con una línea de procesamiento de mensajes fallidos, una base de datos para almacenar dichos mensajes, mejores servicios de monitoreo y métricas más detalladas del funcionamiento de la plataforma.

El desarrollo de Chaski 2.0 fue exitoso y en la actualidad forma parte del proceso productivo de la empresa. Sin embargo, por problemas de planificación, el despliegue de la solución fue problemático para la empresa. Para lograr hacer un despliegue exitoso se desarrolló, en una segunda etapa, una solución que permite que Chaski 2.0 pudiera funcionar en paralelo con Chaski 1.0. Esto fue necesario porque la empresa no podía detener sus servicios para poder desplegar las mejoras del desarrolladas.

Esta memoria busca, además de evidenciar el proceso de desarrollo del software Chaski 2.0, entregar una instancia de discusión acerca de la experiencia de trabajar en una arquitectura de microservicios desplegada en un sistema de Cloud Computing.

Dedicado a Adereso, por darme la oportunidad de trabajar en este interesante proyecto y de formar parte de su equipo. Y a mi familia y amigos por la paciencia y cariño que me han dado. No habría llegado tan lejos sin la ayuda y confianza de las personas que me han acompañado.

Agradecimientos

Agradezco sinceramente al profesor José Alberto Pino por darme indispensable apoyo, sin su ayuda no podría haber entregado este informe.

Quiero también agradecer a mi familia, siempre me han dado auxilio y amor, y lo que soy como persona se los debo a ustedes.

También, no menos importante, agradecer a mis amigos: Macarena, Felipe, Andrea y Ruben. Gracias a todos ustedes por formar parte importante de mi vida. Y gracias al destino que los puso en mi camino.

Finalmente, muchas gracias a Cecilia, mi compañera y amiga, por aguantarme, apoyarme y acompañarme. Solo, no sé donde estaría.

Tabla de Contenido

Índice de Tablas

Índice de Ilustraciones

Capítulo 1

Introducción

Se expondrá en esta tesis el proyecto de desarrollo del mejoramiento del prototipo de repartidor de mensajes Chaski 1.0. El proyecto fue motivado por incitativa privada por encargo de la empresa Adereso. El prototipo Chaski 1.0 fue desarrollado por el autor de la tesis motivado por su práctica profesional II, la que fue realizada durante el verano del año 2019.

Adereso, con el objetivo de mantenerse a la par con los estándares del mercado, requiere que su servicio repartidor de mensajes cumpla con características de calidad de software que no están siendo satisfechas por el prototipo Chaski 1.0. Estos requisitos tienen que ver con la robustez, la persistencia de los datos y la observabilidad del servicio.

Los principales conceptos a desarrollarse en el documento son:

- Desarrollo de software: conjunto de actividades relacionadas con las ciencias de la computación dedicadas al proceso de creación, diseño, despliegue y mantención de software [43].
- Arquitectura *serverless*: modelo de ejecución de código que delega la responsabilidad de la administración de infraestructura a un proveedor de servicios *Cloud* [42].
- Alta disponibilidad: la capacidad de un servicio de funcionar casi en forma completamente continua [52].
- Escalabilidad: la capacidad de un servicio de aumentar o reducir su consumo de recursos en respuesta a cambios en la demanda percibida [33].
- Cobertura y pruebas unitarias: cobertura es una medida de la cantidad de código que está siendo revisado por un test unitario. Prueba unitario es un código cuyo objetivo es comprobar que el software desarrollado cumple con los comportamientos deseados, aislando dichos comportamientos en elementos que pueden ser probados independientemente [55].

Estos temas serán desarrollados con más profundidad en las futuras secciones: estado del arte, problema y solución. De estas, en la sección de estado del arte se discutirán los esquemas arquitectónicos que definieron las decisiones tomadas durante el diseño y desarrollo

de la solución. Asimismo se expondrá sobre las tecnologías usadas y las que se podrían haber usado.

Por su parte la sección de problema se centrará en: describir y el estado y uso de los servicios actuales usados en Adereso para cumplir el objetivo de repartir mensajes internamente en la aplicación Adereso Helpdesk, el servicio legacy basado en procesos pesados y Chaski 1.0, el prototipo desarrollado en la practica; Describir los requisitos que la empresa espera de Chaski 2.0 para poder usarlo cómodamente en su proceso productivo; y en explicar el por qué las carencias de Chaski 1.0 representan un impedimento para que la empresa abandone el uso del servicio legacy repartidor de mensajes.

En la sección de solución se explicará la arquitectura de la solución propuesta, cómo se usaron las tecnologías existentes para lograr cumplir con los requisitos del sistema y se expondrá sobre los problemas para desplegar la primera iteración de Chaski 2.0, y como se superaron estas dificultades.

1.1. Adereso

Adereso es una empresa de desarrollo de software establecida en Chile con equipos de desarrollo presentes tanto en Chile como en México, que ofrece sus servicios internacionalmente en países como Perú, Bolivia y México, además de Chile.

El principal producto ofrecido por la empresa es el Adereso Helpdesk, un software que permite a las empresas que lo contraten concentrar y coordinar gran volumen de conversaciones provenientes de diversos canales, como Facebook, Whatsapp, Twitter y Chat, en objetos denominados tickets, los cuales son atendidos por sus trabajadores. Los clientes de Adereso son empresas con necesidad de organizar sus canales de comunicación, y los usuarios son trabajadores de estas empresas, denominados agentes, quienes son los que efectivamente responden a las conversaciones generadas por los proveedores de mensajes conectados. Además otro producto importante es Adereso Botcenter, servicios relacionados con el desarrollo y montaje de bots.

1.1.1. Clientes de Adereso

Con su producto Helpdesk Adereso ofrece servicios de administración de canales de comunicación a empresas. Para los efectos de la presente tesis e intereses de la compañía, esas empresas se pueden clasificar en dos grupos: “grandes empresas” y “pequeñas y medianas empresas”. Desde el punto de vista de la colocación y comercialización de sus productos, la distinción es importante pues Adereso está concentrando sus esfuerzos en capturar el mercado de las grandes empresas. Este objetivo presenta desafíos tanto comerciales como técnicos para el proceso productivo de la empresa.

Por otro lado, los servicios relacionados a Bots de Adereso son siempre cotizados como proyectos independientes de los otros servicios ofrecidos, por lo que la clasificación no es tan importante en este caso.

Grandes empresas

Según la definición actual en Chile y de acuerdo al criterio de facturación, una gran empresa es aquella que llega a obtener ingresos anuales que superen las 100.000 UF [26]. Sin embargo, aún más importante que la definición normativa, es que para Adereso lo que caracteriza a una gran empresa es el alto volumen de mensajes y usuarios, valores que se acercan a los cientos de miles de mensajes al día y cientos de licencias de usuario. La cualidad de gran empresa es relevante dado que los contratos de estas grandes empresas son negociados por un agente comercial, además tienen derecho a atención prioritaria de soporte técnico por parte de Adereso y, en general, tienen suficiente poder económico como para dirigir el desarrollo del producto en función de sus necesidades particulares, esto reflejado en que se negocian contratos en los cuales estos clientes pueden solicitar nuevas características en el Adereso Helpdesk.

En la actualidad entre los grandes clientes de Adereso en Chile se encuentran: Walmart Chile, Falabella y Mach [2].

Pequeñas y medianas empresas

Como en el apartado anterior y desde el punto de vista de su facturación, las pequeñas y medianas empresas son aquellas cuyos ingresos anuales se encuentran entre las 2.400 UF y 100.000 UF[26]. Sin embargo y como en el caso anterior, desde el punto de vista de Adereso las pequeñas y medianas empresas son aquellas que tienen pocos mensajes y usuarios, considerando que la cantidad de mensajes es muy variable, pero la cantidad de usuarios no supera la decena.

Para este último tipo de clientes se encuentran disponibles planes autogestionados ofrecidos por Adereso. Estos planes limitan entre otros, la cantidad de licencias de usuarios, la cantidad de llamadas a API, la antigüedad de los datos almacenados.

Si bien el segmento de pequeñas y medianas empresas tiene mucha mayor cantidad de clientes potenciales, el esfuerzo por capturar estos negocios es diferente al que ya se ha estado practicando en Adereso, además las necesidades de estos segmentos son diferentes. Es por esto que la empresa ha decidido concentrarse en los contratos ofrecidos por grandes empresas.

Usuarios del Adereso Helpdesk

Por la naturaleza del servicio ofrecido los usuarios del Adereso Helpdesk no son necesariamente los clientes de la empresa. La clasificación de tipos de usuario se realiza en función de los roles que cumplen en la plataforma, una cuenta puede tener varios roles asociados.

Roles:

- Administrador: Tiene el control total de la cuenta, tiene las atribuciones de todos los roles posibles. Sus acciones exclusivas tienen que ver con la conexión y desconexión de

canales de comunicación y la creación de usuarios en la plataforma.

- Analista: Es un rol enfocado en el acceso y extracción de datos sobre el funcionamiento de la plataforma.
- Supervisor: Tiene todas las atribuciones de analista. Adicionalmente puede configurar las reglas de asignación de tickets e interactuar con clientes. Su foco es apoyar con la gestión de la asignación de tickets de los agentes.
- Agentes: Un usuario con rol de agente puede trabajar en el mesón de ayuda. Es el rol que se enfoca en atender a las personas que se comunican con los canales de mensajes administrados a través de Adereso Helpdesk.
- Administrador de equipo: Equivalente a un administrador pero sólo sobre un equipo. Puede crear usuarios y acción total sobre todas las componentes, limitado a los canales y usuarios de sus equipos.
- Supervisor de equipo: Equivalente a un supervisor pero sólo sobre un equipo. Puede acceder a datos, gestionar reglas de asignación y atender tickets, siempre limitado al equipo que tiene a cargo.

1.2. El Problema

El problema que se debe solucionar es esencialmente el de productor/consumidor [54]. Los diversos canales de comunicación conectados al helpdesk producen gran cantidad de mensajes que deben ser enviados al servidor del Helpdesk para ser consumido por los usuarios de la plataforma, el consumidor en este esquema es el servidor del helpdesk. Como Adereso está intentando implementar la aplicación en un patrón de microservicios, es importante que el servicio repartidor de mensajes sea seguro, efectivo y lo más eficiente posible.

Para solucionar este problema en Adereso existe un sistema de procesos pesados y base de datos. Existe un proceso pesado que recibe los eventos de mensajes generados por los diversos canales conectados, este productor ingresa los mensajes en un buffer de comunicación que consiste en una base de datos MySQL. Finalmente el consumidor revisa periódicamente el buffer de comunicación (una base de datos) en busca de nuevos mensajes.

El servicio Chaski 1.0 es un repartidor de mensajes que busca reemplazar el protocolo con procesos pesados. Chaski 1.0 se diseñó para ser escalable y *serverless*, con estas características se busca poder desacoplar el repartidor de mensajes de la infraestructura de los servidores del Adereso Helpdesk. Para cumplir con estos requisitos se usaron servicios de Amazon Web Services, en particular: Amazon Lambda, un producto de funciones como servicio que permite ejecutar código en la red de Amazon; Amazon SQS, un producto de colas de eventos que usaremos como buffer de mensajes; Amazon API Gateway, un producto que permite la publicación de interfaces HTTP.

El prototipo Chaski 1.0 se compone de cuatro componentes principales, la API de entrada, el procesamiento de los eventos de entrada, las colas de mensajes, y un consumidor de la cola que envía los mensajes a su destino. Para usar la interfaz de Chaski el cliente HTTP debe

enviar un objeto Json compuesto de un subject y un body. El subject usado para determinar el paralelismo de las consultas, el body debe contener la url a la que se debe enviar finalmente el mensaje, el contenido de dicho mensaje, el método HTTP a usar y los headers asociados a la consulta final que Chaski debe realizar.

El desarrollo del prototipo por parte del alumno fue un éxito. Chaski 1.0 se comenzó a usar limitadamente por Adereso; sin embargo, este uso limitado está respaldado por el sistema antiguo, los mensajes son procesados por ambos sistemas, pero los consumidores priorizan el Chaski.

1.3. Proyecto

El prototipo Chaski 1.0 fue exitoso, en lo que a un prototipo refiere, cumple con los requisitos funcionales y es un servicio *serverless* y escalable, sin embargo la empresa aún no reemplaza el sistema de procesos pesados por completo.

El principal problema del prototipo Chaski es su falta de robustez. El sistema comunica servicios a través de HTTP y esta conexión puede fallar por diversos temas que el servicio no puede controlar. Estos errores pueden ocurrir porque el servidor remoto no está activo, o lentitud que produce fallos por *timeout*, por ejemplo. En la actualidad si un mensaje no es correctamente enviado el contenido se pierde, y esto es inaceptable.

Otros aspectos del servicio que se deben mejorar antes de que el proyecto sea completamente utilizado por Adereso son: sistema de reintentos de mensajes fallidos; mejor monitoreo y métricas de funcionamiento del servicio. Además, en lo posible también se requiere que exista un sistema de alarmas para detección de problemas con el servicio

1.3.1. Objetivo del proyecto

Mejorar el servicio Chaski 1.0, sistema repartidor de mensajes desarrollado por el autor, para poder ser usado en producción por la empresa Adereso.

1.4. Solución

Se diseñó e implementó una solución que consiste en una línea de procesamiento paralela para mensajes erróneos. De esta forma el servicio ahora consiste de una interfaz HTTP, una cola de mensajes, una función que consume estos mensajes y los intenta enviar a su destino. En caso de que este envío falle se guarda el mensaje en una cola de errores, esta es procesada por otro consumidor que vuelve a intentar enviar el mensaje o guarda el mensaje en una base de datos en caso de ser necesario.

El nuevo sistema responde al problema de mensajes perdidos; cualquier mensaje que no pudo ser enviado exitosamente se almacena, lo que se haga con estos mensajes no estará dentro

de los alcances del proyecto de tesis. El problema de la robustez se enfrenta refactorizando el código para declarar acciones para todos los posibles errores de ejecución. Además, se definieron errores para los mensajes HTTP que agrupa los posibles códigos de respuesta diferentes a 200 (grupo de códigos HTTP 200 es el estándar para consultas exitosas).

Para dar solución a los problemas de monitoreo y métricas se usó el servicio ELK, Elastic Search, Logstash y Kibana de la empresa. Este servicio recibe los logs generados por Chaski y recibe logs diseñados para representar métricas.

Una dificultad encontrada en el desarrollo de la solución es que por la naturaleza independiente de cada una de las funciones lambda se empezó a acumular duplicación de código. Por esto se encapsularon los handlers de los HTTP request, de los logs, el handler que maneja la conexión a la base de datos y el constructor de la interfaz que encapsula el objeto de mensaje. El problema de la duplicación de código también se vio en la implementación de los tests unitarios. Para dar respuesta a la dificultad, los tests se diseñaron como hijos del test general que evalúa las funcionalidades comunes entre las lambdas.

1.5. Alternativas

Los servicios utilizados fueron definidos por el Chief Technology Officer (CTO) de Adereso Sergio Mass en la concepción del prototipo Chaski 1.0 y el uso de estas herramientas formó parte de los requisitos del proyecto. Sin embargo, se puede discutir sobre los motivos de estas decisiones.

El componente más importante en lo que a definir la arquitectura respecta es el buffer de mensajes. El servicio debe poder permitir procesar mensajes en paralelo según corresponda pero al mismo tiempo poder reconocer qué mensajes forman parte de una conversación y por lo tanto deben ser procesados en línea. Las alternativas exploradas fueron RabbitMQ y Kafka; ambas son servicios que deben ser levantados en un servidor manejado por Adereso. La principal característica que llevó a la empresa a optar por Amazon SQS fue que es un servicio completamente serverless administrado por Amazon que cumple con las características necesarias.

Una vez quedó definido que el buffer de mensajes iba a ser implementado con Amazon SQS el resto de la arquitectura del servicio Chaski debía ser implementada con servicios de Amazon: Amazon Lambda, Amazon API Gateway y Dynamodb.

Amazon ofrece un servicio de logs para sus servicios, Amazon Cloudwatch. Este servicio entrega datos del funcionamiento de los productos de Amazon y, en el caso de Amazon Lambda entrega los mensaje definidos en el código. Para poder generar gráficos y alarmas con las métricas de funcionamiento de los productos de Amazon se decidió usar Datadog. Por otro lado, para poder hacer búsqueda sobre el contenido de los logs con el objetivo de poder diagnosticar con mayor facilidad el estado del servicio se decidió enviar los mensajes de registro al cluster ELK (Elastic Search, Logstash y Kibana) de Adereso. Este servicio permite procesar los logs recibidos a través socket TCP y mostrarlos en una plataforma que permite hacer gráficos, streams de logs y búsqueda sobre todos los datos almacenados. También se

decidió usar el servicio ELK para generar métricas a partir de los datos manejados por el servicio Chaski.

Capítulo 2

Estado del Arte

Esta sección del documento se ocupará para indagar sobre tres asuntos. Primero, exploración de dos dicotomías en los paradigmas de arquitecturas de software, monolítico contra microservicios, y *serverless* contra servidores propios. Segundo, revisión del caso de la empresa Netflix y sus decisiones, con especial énfasis en la arquitectura de software. Y tercero, explicación de las tecnologías y herramientas usadas, el proceso de desarrollo de Chaski 2.0, y aquellas herramientas que se descartaron. Las dicotomías a analizar se entenderán estos dos conflictos como independientes, pero en general se asocia a microservicios con tecnologías serverless y al uso de servidores propios con arquitectura monolítica.

2.1. Arquitecturas de Software

2.1.1. Servicio monolítico

En el paradigma de arquitectura monolítica los programas se construyen con una gran base de código que define todos los servicios entregados[3]. En esta base de código se programan las reglas de negocio, las interfaces de usuario, los métodos para acceder a los datos y los modelos de datos. La principal característica de un diseño monolítico es que todas las acciones del software están altamente acopladas, ya sea porque los modelos están relacionados y estas relaciones se deben mantener, o porque existen bloques de código que necesitan ser siempre referenciados por el resto del código.

La arquitectura monolítica da solución a todo el problema de negocio en un solo software, sin embargo, la distinción está en mayor relación a cómo se administra una aplicación, no respecto de la calidad de su código. Un servicio monolítico puede ser diseñado con alta modularidad, pero el servicio está pensado para ser levantado en su totalidad.

2.1.2. Microservicios

La arquitectura de microservicios se refiere a la forma de entender una aplicación como una red de servicios autónomos e interconectados, en la cual cada servicio se encarga de satisfacer una característica específica en el contexto general del software[45] que se está desarrollando. Cada uno de estos microservicios se puede desplegar independientemente, lo que da mucha flexibilidad al momento de repartir recursos en la mantención de la aplicación publicada.

Cada uno de los microservicios debe cumplir, en general, con las siguientes características:

- Tener su ambiente de ejecución completamente definido independiente del ambiente de otros servicios[28].
- Definir un protocolo de comunicación entre los diferentes servicios[28].
- Resolver en su totalidad una necesidad específica y desacoplada del problema de negocio de la aplicación global[28].

2.1.3. Software serverless

La arquitectura *serverless* es un paradigma de diseño de software basado en *cloud computing*, para no hacer consideraciones sobre el tipo de decisiones que se pueden tomar al hacer el diseño de un software basado en servicios alojados en la nube, hablaremos de *serverless* como una definición más específica de *cloud computing*.

Cloud computing, definido por el The National Institute of Standards and Technology (US), como un modelo de computación que permite acceso ubicuo, conveniente y *on-demand* a una red compartida de recursos de computación (como redes, servidores, almacenamiento, aplicaciones y servicios). Este acceso puede ser rápidamente aprovisionado y entregado sin mucho esfuerzo o interacción con el proveedor de servicios[44].

El concepto de arquitectura de software *serverless* no implica que no exista un servidor y hardware corriendo el software, implica que el ingeniero no tiene que preocuparse por dicho servidor. La arquitectura *serverless* ha permitido a los equipos de desarrollo definir un software en términos de flujos de datos, funciones lógicas distribuidas y bases de datos altamente administradas[59], donde el proveedor se encarga de solucionar problemas de hardware e infraestructura.

En el desarrollo de un software existen muchos problemas que se deben enfrentar, además de la función de producción o necesidad que se desea resolver. La arquitectura *serverless* permite a los desarrolladores desviar la atención de tres de estas necesidades no funcionales; la infraestructura, el aprovisionamiento, y escalabilidad[59].

Existen dos principales formas de usar la arquitectura *serverless*, BAAS (Backend as a Service) y FAAS (Function as a Service). En BAAS el proveedor de servicios cloud entrega a los desarrolladores una plataforma donde puede alojar sus servidores, y un sistema de red

que permite publicar sus servicios. FAAS es un patrón en el que se implementan variadas funciones atómicas que toman los datos y los procesan para ser tomados por otras funciones. Implica en general[58] un diseño stateless de las funciones, pues una instancia solo se puede comunicar con otra o la base de datos a través de eventos I/O.

2.1.4. Levantamiento de servidores manejados

La forma clásica de entender la arquitectura de software considerando los servidores manejados es el self-hosting. Sin embargo, se entiende como servidores manejados todo ambiente de ejecución donde el equipo de desarrollo debe encargarse de solucionar los problemas relacionados con el levantamiento de servicios, sin descartar que estos servidores pueden ser alojados en un servidor ofrecido por un proveedor de servicios de *cloud computing*.

Se pueden resumir los problemas a resolver para poder levantar bien un servicio en un servidor en:

- Identificar y satisfacer los requisitos de hardware que necesita el servicio a desplegar[53].
- Implementar un protocolo de red para publicar el servicio en la medida que sea necesario para satisfacer el problema de negocio[53].
- Identificar y satisfacer requisitos de seguridad de las máquinas en las que se despliega el servicio[53].
- Identificar y satisfacer los requisitos de software que necesita el servicio a desplegar[53].

Este diseño se debe pensar no sólo al momento de levantar el servicio, sino que también se debe tener en cuenta en futuros despliegues de nuevo código y funcionalidades.

2.2. Caso Netflix

Netflix es una empresa de distribución de contenido audiovisual con operaciones a nivel mundial[50]. Se consideró relevante mencionar esta empresa pues ha publicado considerable material referente a su proceso de modernización, centrado en implementar una arquitectura de microservicios con una infraestructura montada con servicios de *cloud computing* de Amazon Web Services[15]. En particular se citará un par de publicaciones en el blog de Netflix referentes a la implementación y creación de herramientas para facilitar la administración de una aplicación como Netflix[21][32].

Se describe la experiencia de desarrollar código en Netflix como una muy similar a aquella obtenida de trabajar en plataformas *serverless* e incluso FAAS[21]. Una de las características mas importantes para la empresa de la nueva forma de entender la aplicación y de implementar la infraestructura tiene que ver con el rendimiento y comodidad de sus trabajadores al momento de desarrollar código. Los desarrolladores sólo se deben hacer responsables del

adaptador que están creando, y no se tienen que preocupar de los problemas de infraestructura o administración de servidores que pueden generarse[21].

Todo sistema necesita ser administrado y todo patrón que se use para diseñar el sistema tendrá sus beneficios y problemas particulares. Netflix explica que si bien la experiencia de los desarrolladores se ve positivamente impactada, el levantamiento y mantención de los servicios tiene sus propios problemas. "En *serverless*, una combinación de unidades de despliegue más pequeñas y con una mayor abstracción, generan importantes beneficios tales como: aumento de velocidad y mayor escalabilidad, menores costos, y en general la habilidad de poder enfocar los esfuerzos en nuevas características del producto. Sin embargo los problemas operacionales no se eliminan, sólo toman una nueva forma o incluso aumentan. Herramientas de control operacional deben desarrollarse para responder a estos nuevos desafíos"[21].

Los desafíos descritos por Netflix en las publicaciones citadas se agrupan en: desarrollo, versionamiento, pruebas unitarias, modularidad del código, despliegues, observabilidad y administración del servicio[21][32].

El desarrollo en una plataforma *serverless* permite resolver los problemas de forma mucho más acotada, entendiendo segmentos más limitados de código. Sin embargo, este beneficio conlleva sus limitaciones. La principal limitación encontrada en la experiencia de netflix se produce por la inhabilidad de evaluar los cambios en la plataforma localmente. Por la naturaleza distribuida y desacoplada de todos los servicios que componen el producto, los cambios realizados por los trabajadores no pueden ser probados localmente[21]. Otro factor interesante que reporta la empresa, es la emergencia de patrones negativos en sus desarrolladores, en particular el uso de logs muy verbosos para evitar tener que realizar nuevas iteraciones del proceso de desarrollo[21].

La necesidad de llevar un buen registro de la versión de la aplicación se transforma en un interesante problema cuando la aplicación funciona en un patrón de micro servicios. Cada uno de los servicios que conforman el producto llevan sus propias versiones, y estas no tienen por qué ser compatibles entre sí. Netflix notó en particular que cada equipo implementa sus propias políticas de versionamiento. Por esto se implementó un patrón donde el significado semántico de la versión se pierde y todo lo que importa es la implementación del software versionada en un artefacto ejecutable[21].

El patrón de *cloud computing* con arquitectura de microservicios genera un ambiente en el cual las pruebas unitarias y la integración continua tienen que ser diseñados para replicar en la mayor medida posible el contexto en el que la aplicación final estará funcionando. En particular, el patrón remoto de ejecución implica que las pruebas de integración sean de alta complejidad. En Netflix identifican tres diferencias fundamentales entre sus ambientes de prueba y de producción: el volumen de tráfico, el tiempo de vida de las instancias y la viabilidad de las pruebas unitarias y de integración[21].

El tema de la modularidad del código es un problema muy de la mano con el versionamiento. Cuando la cantidad de microservicios crece, la necesidad de definir patrones para el re-uso de bloques de código comunes aumenta, sin embargo, la dificultad en el control de las versiones y compatibilidad de estos bloques también crece. Netflix desarrolló un patrón para definir sus librerías compartidas siguiendo el patrón de versionamiento, que permite a

sus artefactos de despliegue poder determinar qué versión de librerías necesita y con qué versiones se puede comunicar[21].

El despliegue de plataformas *serverless* tiene sus propias complicaciones, algunas heredadas del proveedor, otras de la naturaleza del proceso. En particular, Netflix relata que las funcionalidades nuevas son particularmente susceptibles a "*cold starts*"[32]. El tráfico de consultas no es suficiente para que la instancia de ejecución se calibre para responder a las necesidades de servicio que se desea prestar, por ejemplo en la cantidad de invocaciones que se necesitan. Para remediar la situación, en Netflix se desarrollaron aplicaciones que se encargan de levantar los nuevos servicios y darles un procesamiento de "calentamiento"[32]. Además, la empresa destaca la utilidad del despliegue por canary y las pruebas con múltiples variables en el despliegue de nuevo código[32]. El despliegue por canary corresponde a un patrón de entrega de nueva versión de una aplicación a un porcentaje creciente de los usuarios, partiendo con un porcentaje muy bajo y aumentando poco a poco hasta que se libera la nueva versión de la aplicación a todos los usuarios[9].

El problema de monitorear una aplicación basada en microservicios es el volumen de la información; cada instancia de ejecución de cada microservicio genera datos de bajo nivel que pueden ser monitoreados. Esto se traduce en que, por ejemplo, las alarmas de nivel de uso de una máquina están constantemente sonando, pues al menos una de las instancias puede estar en sobreuso de sus recursos. Por esto, Netflix destaca que todas las métricas de bajo nivel que están generando las instancias deben estar siempre viéndose reflejadas en métricas de alto nivel que entreguen una visión compuesta del estado de salud de la aplicación[32].

La administración de la aplicación una vez que ya está funcionando es un problema de todo proyecto de software, sin embargo, el patrón de arquitectura *serverless* trae consigo problemas particulares. El determinar qué versión del código está corriendo en qué instancia y qué clientes están consultando se transforma en un problema de alta complejidad, sobre todo cuando la cantidad de versiones e instancias tiende a aumentar. En este contexto Netflix comparte sus experiencias, comentando que pueden ocurrir errores humanos a la hora de levantar nuevo código o dar de baja instancias, provocando problemas a los usuarios que están en ese minuto usando la aplicación[32]. La empresa nos dice: "*serverless* hace fácil el desplegar aplicaciones y olvidarse del proceso de despliegue, pero esto trae consigo un aumento en las consideraciones de mantención de dicha aplicación"[32].

2.3. Tecnologías y servicios

2.3.1. RabbitMQ

RabbitMQ es un repartidor de mensajes *open source*, escrito en earlang. Se le reconoce ser simple, fácil de desplegar, y con opciones tanto para funcionar en un patrón distribuido como federado de alta disponibilidad[57]. RabbitMQ soporta los protocolos de transmisión de mensajes STOMP, MQTT y AMQP, además de HTTP como wrapper de alguno de los tres protocolos ya nombrados.

STOMP es un protocolo de transmisión de mensajes centrado en contenido de texto que busca ser simple y fácil de implementar[60]. MQTT es un protocolo de mensajes centrado en contenido binario especialmente orientado a ser muy eficiente[48]. AMQP es un protocolo de transmisión de mensajes centrado en la seguridad y la confianza de la comunicación[51].

En lo que al proyecto Chaski respecta existen dos factores que determinaron que RabbitMQ no era el producto adecuado para satisfacer la necesidad de la empresa:

1. Si bien RabbitMQ es simple de instalar siguiendo la guía, si se quiere personalizar la configuración para satisfacer de la forma más eficiente la necesidad de la empresa, se debe desarrollar un archivo de configuración de más de 50 potenciales variables[56], no se busca sugerir que se necesiten usar todas estas variables, pero para estar seguros de que se tiene la mejor configuración posible el dominio del software por parte del equipo debe ser extenso.
2. Si bien se puede usar RabbitMQ de forma distribuida para asegurar su alta disponibilidad, el trabajo de redes, el monitoreo del cluster y la mantención de todo el sistema no son triviales, esto significa que para mantener Chaski en producción con RabbitMQ se debe considerar el costo humano de mantener los servidores.

2.3.2. Kafka

Apache Kafka es un software *open source*, desarrollado en java y scala, centrado en un alto flujo constante de datos[19] que funciona como un paradigma de logs solo de escritura[49], en el cual el productor de mensaje agrega información al centro de datos para que el cliente decida desde dónde y cuánto leer. Kafka usa su propio protocolo de mensajes[20].

Kafka tiene como objetivo dar solución al flujo de mensajes bajo un paradigma de publicador y suscriptor[49], con soporte para múltiples productores y subscriptores.

Los motivos por los cuales Adereso decidió no usar RabbitMQ también se aplican para Apache Kafka. Sin embargo, el caso de RabbitMQ se diferencia del de Kafka en que el segundo *broker* de mensajes es una herramienta más compleja de lo necesario para dar solución al problema de la empresa. Adaptar el problema de *broker* de mensajes desde el patrón productor-consumidor a un patrón de publicador-subscriptor no es altamente complejo, pero las ramificaciones de esta adaptación podrían llevar a dificultades no predecibles, y completamente innecesarias.

2.3.3. Celery

Celery es un administrador de colas de tareas, su input corresponde a una unidad de trabajo. Celery administra la ejecución de las tareas monitoreando las colas para ejecutar las unidades de trabajo según corresponda. Celery permite mantener múltiples trabajadores y organizadores para permitir un sistema de alta disponibilidad y escalable[27].

En Adereso, Celery se usa para varias tareas, pero en lo que al proyecto respecta Celery es el componente que se encarga de encolar las escrituras del buffer (una base de datos) entre los productores de mensajes y los consumidores para el sistema de procesos pesados que resuelve el problema para repartir mensajes en Adereso Helpdesk.

2.3.4. Servicios de Amazon

Amazon SQS

Amazon SQS es un servicio de colas de mensajes completamente administrado que ofrece una plataforma para definir el envío, almacenaje y recepción de mensajes sin pérdida de los datos entregados[7]. Se presenta a AWS SQS como una forma simplificada de implementar un sistema de comunicación entre servicios de software, con alta disponibilidad y cuya escalabilidad está asegurada por la red de *cloud computing* de Amazon Web Services. Se puede usar AWS SQS en dos tipos de colas, estándar y FIFO[13]. De manera simplificada se puede entender SQS como un software de caja negra con dos interfaces, una para generar eventos de mensajes que se deben guardar en las colas y una de salida, que genera los eventos de mensajes que pueden ser consumidos por otro servicio.

Las colas estándar de SQS son best effort en lo que al orden de entrada de los mensajes refiere, aseguran que el evento de salida de la cola ocurre al menos una vez por cada mensaje y tienen una tasa de tráfico de mensajes ilimitada[13].

Las colas FIFO de SQS ofrecen una tasa de flujo de mensajes limitado a 300 eventos de entrada o salida de la cola por minuto. Sin embargo, aseguran que el orden de los eventos de salida es el mismo de los eventos de entrada y también que por cada mensaje que ingresó a la cola sólo se levanta una vez el evento de salida[13]. Gracias a esta característica fue que se decidió realizar el proyecto Chaski con Amazon SQS, y Amazon en general. La funcionalidad compleja del proyecto es asegurar el orden de los mensajes y que el servicio que permite encolar los eventos fuera escalable y de alta disponibilidad. Gracias a Amazon SQS se dispone de una solución fácil de implementar que entregaba todas las características necesarias.

Amazon Lambda

Amazon Lambda es un servicio de computación en la nube que permite ejecutar código desplegado como un paquete de implementación en zip o imagen de contenedor[8]. Es un ejemplo de producto diseñado para *cloud computing* en paradigma FAAS, y es el servicio usado para implementar todo el código fuente del proyecto Chaski. Lambda es un servicio puramente stateless, esto permite a la infraestructura lanzar tantas instancias de ejecución como sean necesarias[12], limitado por configuraciones del proyecto.

Amazon Lambda es un servicio que se cobra por consumo considerando la cantidad de invocaciones, la memoria requerida y el tiempo de ejecución de las instancias[18]. Y dentro de las características del servicio las más relevantes para el proyecto son: escalabilidad automática, conectividad con los otros productos de la red de Amazon y la disponibilidad de

ejecución provisionada[12]. Una limitación de lambda es que solo admite ciertos lenguajes: Java, Go, PowerShell, Node.js, C#, Python, y Ruby[12].

Amazon API Gateway

Amazon API Gateway es un servicio para publicar interfaces de API web simplificado y administrado por la red de servicios de Amazon[4]. API Gateway ofrece dos patrones de conectividad, HTTP API y Websocket[10], este último tiene la particularidad de ofrecer mantener la conexión con el cliente para mantener conexión bidireccional.

API Gateway tiene un patrón de cobros basado en la cantidad de llamadas y los datos transferidos por estas[16]. Ofrece una forma simple de manejar permisos de acceso, seguridad, balanceo de tráfico y versionamiento de la API[10]. Para el proyecto Chaski lo más importante es que usar API Gateway es la forma más simple y directa de ofrecer los servicios encapsulados en las funciones lambdas al backend de Adereso Helpdesk. Esto implica que podrían hacer llamadas directas a las instancias de Amazon Lambda desde el servidor de Adereso, pero los problemas de seguridad y permisos que esto significa complican innecesariamente la publicación del proyecto.

Amazon DynamoDB

DynamoDB es un servicio de base de datos no relacional, guarda la información en documentos tipo json, modelo de llave y valor[6]. Las bases de datos en Dynamodb pueden tolerar cantidades de más de diez trillones de consultas por día con picos de carga mayores a veinte millones de requerimientos por minuto[11].

Por su naturaleza documental dynamodb es una base de datos flexible con respecto a los campos y los valores, la única restricción es que todos los documentos deben tener una llave primaria y todas las llaves primarias deben ser diferentes[14].

El patrón de precios de Dynamodb es algo complejo, se cobra por el tamaño de la base de datos, cantidad de operaciones de lectura-escritura y el volumen de datos que se mueven las consultas[17]. Pero aun así es un patrón de cobro por uso.

Amazon CloudWatch

CloudWatch es un servicio de monitoreo y observabilidad de la plataforma de servicios de Amazon. Recolecta información de logs, métricas y el funcionamiento general de los recursos usados[5]. Amazon recolecta automáticamente los mensajes de log de las instancias que están siendo ejecutadas, pero además registra informes de rendimiento de las instancias y los recursos en general, como por ejemplo, tiempo de ejecución, memoria utilizada y cantidad de instancias siendo ejecutadas.

Además CloudWatch también ofrece soluciones de visualización de sus datos, con dashboards por servicio y una herramienta de búsqueda para facilitar el análisis de los registros

de log almacenados en CloudWatch[5].

AWS CloudWatch fue una importante herramienta durante el desarrollo, monitoreo y depuración del proyecto Chaski. Pero su uso es muy técnico y limitado a los servicios de Amazon. Por esto se optó por herramientas externas que pueden concentrar la información de múltiples servicios.

2.3.5. Servicios de Google

En esta sección se expondrá sobre el estudio de los servicios de Google, es importante mencionar que esta investigación se realizo pese a que el uso de servicios de Amazon ya estaba decidido. Los servicios de Google fueron descartados para el desarrollo del proyecto porque existe SQS en la red de servicios de Amazon, y Google Cloud no tiene una alternativa que haga lo mismo, de la misma forma.

CloudRun

CloudRun es un servicio de despliegue de contenedores, se encarga del manejo del servidor y entrega interfaces de red incluyendo una dirección pública por defecto que puede ser usada desde internet[36]. CloudRun es un ejemplo de cómo usar la arquitectura de *cloud computing* en el patrón de BAAS, con la particularidad de funcionar con tecnología de contenedores. Permite montar un servidor sin preocuparse de los temas de hardware o red. Uno de los beneficios de usar un servicio como CloudRun es que permite total libertad con respecto al lenguaje y estructura de la aplicación que se quiere implementar.

El cobro del servicio CloudRun se basa en el tiempo de cpu usado por la aplicación desplegada, la memoria destinada al servicio y la cantidad de consultas realizadas al servidor[37].

En lo que a Chaski respecta, un servicio como CloudRun no era tan interesante; se diseñó y planificó para funcionar en base a un patrón FAAS, por su simpleza. Sin embargo, no es difícil imaginar a Chaski como un servidor cuyos endpoints realizan el trabajo de las distintas lambdas.

Cloud Functions

Google Cloud Functions es un servicio de computación en la nube que permite levantar procesos atómicos y stateless[40], siguiendo el patrón de desarrollo serverless FASS. El producto entrega un entorno de ejecución para el código desplegado; dicho código puede ser escrito en JavaScript, Python, Go o Java[39]. Google se encarga de montar toda la infraestructura para que el código pueda ejecutarse y ofrece alta disponibilidad y escalamiento automático del servicio desplegado[39]. Cloud Functions se cobra por uso considerando el tiempo de ejecución, la cantidad de instancias y la cantidad de recursos ocupados por las instancias[35].

En lo que al diseño del proyecto respecta, Amazon Lambda y Google Cloud functions son, sin considerar el resto de los productos ofrecidos por cada proveedor, intercambiables. La única diferencia relevante entre estos productos es la lista de lenguajes soportados, pero como Chaski fue implementando en Python esto tampoco es relevante. La decisión, al diseñar el proyecto, fue tomada considerando todos los productos relevantes ofrecidos por el proveedor.

Cloud SQL

Cloud SQL es un servicio de bases de datos administrados ofrecido en la red de recursos cloud de Google[38]. Toda la infraestructura necesaria para montar un servidor de bases de datos es manejada por Google y entregan una interfaz para definir usuarios y permisos, conexiones al servicio, seguridad, disponibilidad y escalabilidad[38]. El patrón de precios de Cloud SQL, en general, se cobra por el tipo de máquina en la cual se aloja la base de datos, la cantidad de memoria necesaria y la cantidad de consultas[41].

Desde el punto de vista de Chaski, Cloud SQL es un servicio más completo y complejo de lo necesario, ya que sólo se necesita una base de datos donde guardar los registros de los mensajes. Por eso, sin considerar el resto de los servicios de cada red, Cloud SQL no habría sido la primera opción para el proyecto.

2.3.6. Servicios de monitoreo

Datadog

Datadog es un software bajo el patrón de venta SaaS, que permite concentrar y visualizar métricas de diferentes fuentes de datos, en particular de servicios de Google Cloud y de Amazon Web Services[25]. En relación con el proyecto Chaski, la principal característica es su facilidad para conectarse a las diversas fuentes de datos[24]. Para Amazon, por ejemplo, se debe configurar una cuenta de servicio y posterior a eso Datadog se encarga de discriminar qué datos pueden obtenerse desde Amazon. También resultaron importante las herramientas de dashboards y visualizaciones[24]; con eso se generan vistas para monitorear diversas métricas de diversos servicios. Finalmente también son usadas las alarmas y la integración con Slack ofrecida por Datadog.

Datadog fue la primera opción para satisfacer las necesidades de monitoreo del servicio Chaski, pero no es la única herramienta usada, se usan también los servicios de CloudWatch y del servidor ELK de Adereso. La principal falencia de Datadog en relacion al proyecto es su patrón de cobro y la dificultad, y limitaciones, para definir métricas personalizadas.

ELK

ELK es el nombre común que se da al uso concertado de tres servicios de Elastic, Elasticsearch, Logstash y Kibana[30]. Elasticsearch constituye un motor de búsqueda, Logstash es una interfaz altamente escalable que recibe datos y los almacena en algún lado, y Kibana es

una aplicación para explorar y visualizar información almacenada en alguna base de datos de Elasticsearch[30].

Elasticsearch es un motor de búsqueda no relacional basado en documentos que se usa de forma muy similar a una base de datos, ofrece una interfaz *restfull* para acceder y editar los datos[47]. Adereso tiene un servidor de Elasticsearch dedicado a manejar registros de funcionamiento y métricas de los servicios. La funcionalidad más importante para la empresa es la búsqueda textual dinámica, la que se usada desde Kibana para analizar los datos de registros generados.

Logstash es un concentrador de datos, múltiples fuentes pueden invocar logstash para que este servicio transforme el input y lo envía a un destino específico, ofrece múltiples interfaces de acceso todas sobre HTTP[29]. Adereso tiene un cluster de servidores de Logstash que envían sus datos al servicio Elasticsearch de la empresa. Lo más importante es el formateo de datos desde la fuente de datos, permitiendo por ejemplo extraer mensajes json embebidos dentro del texto que se recibe. Además Logshtash se encarga, configurado por Adereso, de definir los índices de datos de Elasticsearch donde se ingresan los mensajes recibidos por el servicio.

Kibana es una aplicación que ofrece un frontend para visualizar los datos almacenados en una base de datos elasticsearch. Ofrece herramientas para monitorear el servicio Elasticsearch usado, además de herramientas para explorar y visualizar los datos contenidos en la base de datos[31]. Adereso cuenta con un servidor de Kibana disponible para su uso por el equipo de desarrollo y el de soporte, con visualizaciones y dashboards configurados.

2.4. Metodología de desarrollo

2.4.1. Testing

Las pruebas de software son una forma en la que el equipo puede asegurar que el software desarrollado es correcto, de alta calidad y de buen rendimiento. En general ayudan a una organización asegurando que el software se comporta como se espera[22]. En particular las pruebas unitarias son la unidad más básica de pruebas de software, con esta práctica se busca asegurar que pequeños bloques de código, denominados unidades, se comportan como se espera ante input entregando los resultados correctos[22].

2.4.2. Coverage

El coverage es una medida de la proporción del proyecto que fue evaluado en alguna prueba, es una medida de la calidad del conjunto de pruebas en el proyecto[55]. Existen cinco medidas de coverage: de funciones, de declaraciones, de ramas, de condiciones y de líneas[55]. De estas medidas la que importa para el proyecto Chaski es la de líneas, se busca con ello que la mayor cantidad de líneas sean ejecutadas en las pruebas.

2.4.3. Pull Requests

Pull request es una herramienta de los productos de control de versionamiento, bitbucket en el caso de Adereso, que permite compartir los cambios realizados al código con los compañeros[34]. Es una estructura que agrupa los cambios del código, explicaciones del diseño implementado y da espacio para que otras personas puedan comentar sobre el código línea por línea de ser necesario[34]. En Adereso se usa como una herramienta para mejorar la calidad del código realizado, dando espacio a la crítica por parte de los pares y una forma de compartir el conocimiento. Al momento de revisar el pull requests el desarrollador debería poder entender el porqué y el cómo de los cambios realizados por el compañero.

2.4.4. Integración Continua

La integración continua es el conjunto de procesos que buscan automatizar la integración del código desarrollado por muchos desarrolladores en un mismo proyecto con el objetivo de permitir una continua unión del código realizado al repositorio general[23]. El proceso consiste en desplegar un ambiente con los cambios y correr todos las pruebas del proyecto. Si las pruebas son exitosas se pueden unir los cambios al código fuente. En adereso el CI se usa para rendir cuenta en la página de pull request del éxito de los test que fueron realizados automáticamente.

2.4.5. Control de calidad

El control de calidad es el conjunto de métodos, herramientas y técnicas que permiten asegurar la calidad del producto. En Adereso se cuenta un equipo de control de calidad que se encarga de comprobar que cualquier cambio aprobado por el equipo no afecte el flujo normal de Adereso Helpdesk de formas inesperadas. Las pruebas de QA se hacen en un ambiente similar al de producción.

Capítulo 3

Problema

El desarrollo de Chaski 1.0 en el contexto de la práctica del alumno fue un éxito, ya que se cumplió con los requisitos del proyecto. Sin embargo, para poder usar cómodamente el nuevo servicio Adereso necesita mejoras fundamentales. Estas mejoras están enfocadas en aumentar la robustez y la calidad del monitoreo del servicio.

En la actualidad conviven dos protocolos de comunicación entre los productores de mensajes y los servidores de la aplicación Helpdesk en Adereso. Un sistema de servidores manejados por Adereso que reciben eventos desde los proveedores y se comunican con las intancias de Adereso Helpdesk escribiendo en una base de datos, la cual es constantemente revisada dichas instancias en busca de nuevos mensajes. Y el proceso que usa Chaski para conectar los servicios que generan mensajes y los que deben consumirlos.

El servicio que usa procesos pesados y una base de datos para comunicar los generadores de mensajes con los consumidores tiene complicaciones importantes que limitan la escalabilidad del servicio, sin embargo, lleva años funcionando. Para migrar a usar unicamente Chaski 1.0 se debe dar seguridad a la empresa de que su proceso productivo no se verá afectado negativamente, y más aún, de que los beneficios valen la incertidumbre del nuevo desarrollo.

3.1. Situación actual: servicio legacy

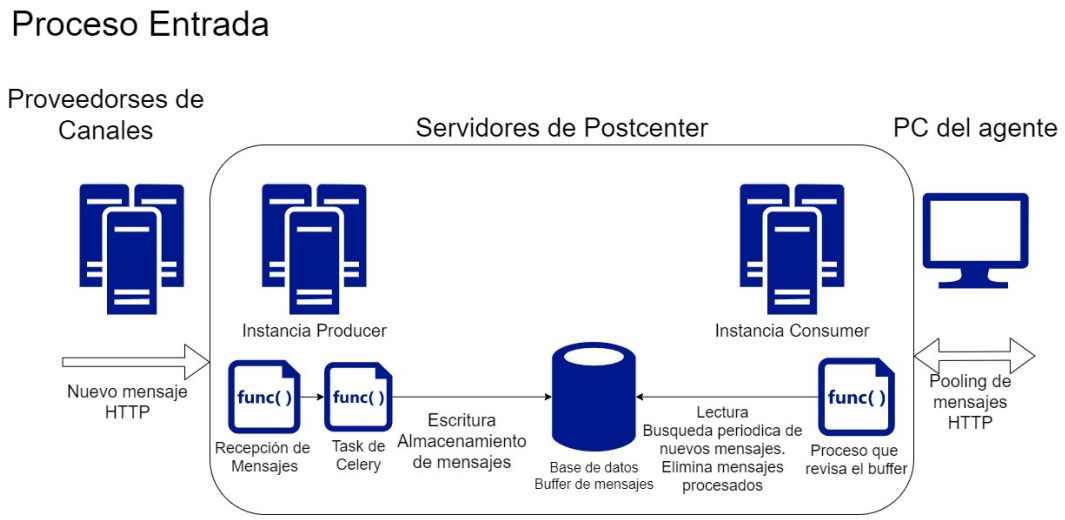


Figura 3.1: Esquema general del sistema de procesos pesados para manejar mensajes provenientes desde proveedores de canales de comunicación.

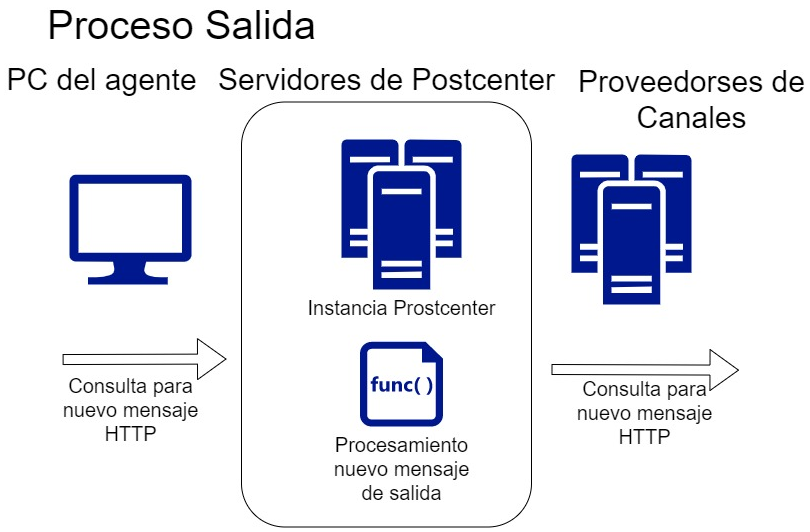


Figura 3.2: Esquema general de cómo las instancias de Helpdesk envían mensajes a los proveedores de canales de comunicación.

3.1.1. Arquitectura del Servicio

La solución actual que en la empresa Adereso se usa para satisfacer la necesidad de comunicación entre los productores de mensajes y las instancias de ejecución de Adereso Helpdesk corresponde a un subconjunto de las instancias de la aplicación que solo se encargan

de responder al endpoint externo de mensajes que se entrega a los proveedores de canales de comunicación. Estas máquinas, una vez recibido un mensaje encolan el mensaje en una tarea de celery, esta tarea se encarga de escribir en la base de datos (el buffer de mensajes). Por su lado las instancias de Adereso Helpdesk que se encargan de manejar las interacciones con los usuarios leen periódicamente la base de datos usada como acumulador de mensajes en búsqueda de nuevos eventos.

Un detalle relevante de este sistema que no fue descrito es que hay varias máquinas encargadas de hacer la función de productor de mensajes en el esquema, y esta separación se hace por clientes, el cliente A, B y C por ejemplo se manejan en la máquina 1, el D y el E se manejan en la máquina 2. Pero el buffer de mensajes es único para todas las instancias de productores y consumidores, lo que genera un gran cuello de botella.

3.1.2. Problemas de escalabilidad

Existen dos aristas en el problema de escalabilidad de la solución legacy que se está usando en Adereso. Por un lado, en periodos de baja carga, por ejemplo a la 1 am en Chile, el servicio no puede consumir menos recursos, el hardware en el que se montó el servicio no se puede reducir automáticamente. La otra situación que se puede generar es que se sobrepase la capacidad del servicio. Por la manera en que está construido el servicio repartidor de mensajes, la escalabilidad podría realizarse verticalmente y aumentar los recursos de las máquinas que estén sobrecargadas, pero esto se tendría que hacer por cada instancia.

En la medida que la demanda por el producto crece el problema de la escalabilidad se hace cada vez más urgente. Desde el punto de vista de los recursos, llega un punto en que cada hora que el servicio no se usa en la medida justa, la empresa incurre en costos importantes y evitables. Y desde el punto de vista de la calidad del servicio, si el flujo de mensajes se detiene porque el servicio repartidor de mensajes llegó al tope de su capacidad los clientes verán negativamente afectado su servicio.

3.1.3. Problemas de mantención

El servicio repartidor de mensajes legacy está completamente acoplado a las instancias de ejecución de Adereso Helpdesk, de hecho, su código forma parte de la base de código del proyecto completo, monolíticamente. Esta situación genera dos problemas importantes, primero, cambios en la funcionalidad de Adereso Helpdesk podrían afectar de manera inesperada al servicio repartidor de mensajes, y una caída del servicio podría tener muchos puntos de falla, todos fatales. Segundo, como desarrollador navegar el código fuente del proyecto para entender cómo funciona el servicio repartidor de mensajes es una tarea de alta complejidad.

Otro problema de mantención es que el servicio tiene cuellos de botella muy graves, en particular la base de datos que se usa como buffer. Cualquier error en este servicio genera una caída total de la comunicación, es solo una instancia de ejecución para todos los mensajes.

3.2. Situación actual: prototipo Chaski

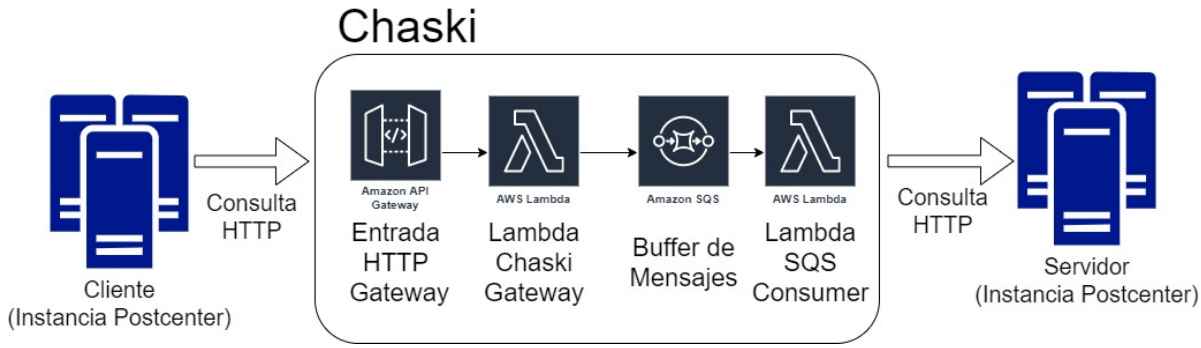


Figura 3.3: Esquema general de diseño del prototipo Chaski.

3.2.1. Arquitectura del Servicio

El prototipo Chaski se compone de cuatro módulos fundamentales: Una API de entrada para ofrecer y controlar el acceso al servicio implementada con AWS API Gateway; Una función lambda que recibe los eventos generados por la API de entrada, cuya responsabilidad es determinar la correctitud de la consulta enviada al servicio y encolar el mensaje según corresponda; Una serie de colas First In First Out implementadas en AWS SQS; Y una función lambda que recibe eventos generados por la cola SQS cuya responsabilidad es realizar la consulta HTTP con el mensaje al servidor objetivo. Ambas funciones lambdas, lo que representa la base del código fuente del proyecto, están escritas en Python 3.6.

Adicionalmente a los módulos principales, Chaski hace uso de servicios de monitoreo de Amazon, el servicio de stream de logs Cloudwatch, y una integración con Datadog para monitoreo más detallado y centralizado del servicio completo.

Finalmente, para el proceso de integración continua se usa Drone CI, el final del flujo de Drone CI es la generación de un paquete de implementación zip de las funciones lambdas que se almacena en S3 de AWS.

3.2.2. Uso actual del Chaski

El servicio está siendo usado limitadamente como intermediario de mensajes directos de canales de comunicación y como repartidor de mensajes para funcionalidades internas. En este contexto el servicio maneja en promedio veintisiete mil mensajes por día. Y los tiempos de ejecución de estos mensajes treinta milisegundos para API Gateway y de dos segundos para el SQS consumen, en promedio. Valores observados durante el mes de enero del 2020.

El prototipo Chaski se usa conservando el sistema antiguo, basado en máquinas y bases de datos, como respaldo del servicio. Esto es inaceptable pues Chaski se pensó como un reemplazo de la funcionalidad no como un servicio paralelo.

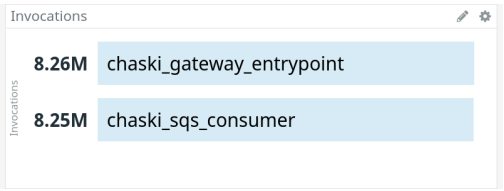


Figura 3.4: Foto total invocaciones durante el mes de enero del 2021 de la lambda Chaski Api Gatway y SQS Consumer.

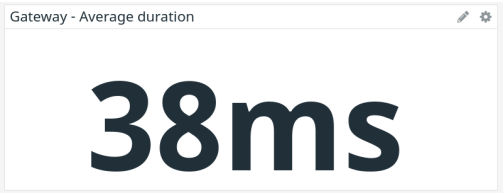


Figura 3.5: Foto promedio duración invocación de la lambda Chaski Api Gateway durante el mes de enero del 2021.

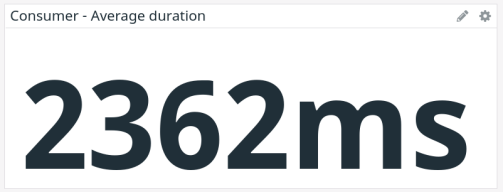


Figura 3.6: Foto promedio duración invocación de la lambda Chaski SQS Consumer durante el mes de enero del 2021.

3.2.3. Monitoreo

Figura 3.7: Foto vista general del monitoreo en Datadog de Chaski durante enero del 2021

Amazon maneja datos de monitoreo de sus servicios internamente, se usan activamente los gráficos de actividad de Lambda y los streams de logs concentrados en cloudwatch, también

con datos generados por lamda.

También existe una integración con datadog para generar dashboards de gráficos y datos más descriptivos del servicio Chaski en plenitud, se observan datos relevantes tanto de las lambdas como de las colas sqs, todos en el mismo dashboard.

3.2.4. Errores observados

Chaski es un servicio que se desempeña en un contexto incierto, debe ser capaz de definir cuáles de los mensajes recibidos deben ser procesados y cómo, y el punto culmine de las labores del servicio es hacer una consulta HTTP a un servidor remoto.

Para el punto de entrada el potencial de levantar errores es limitado, se diseñó el sistema pensando que el cliente que solicite los servicios de Chaski se va a encargar de realizar una consulta válida, por lo tanto podemos limitar el dominio que debe ser correctamente manejado por la lambda API Gateway. Otra fuente de errores de la API Gateway es la inserción de mensajes en la cola SQS, sin embargo, si esto falla todo el sistema se cae, y no hay mucho en términos de recuperación posible.

Por otro lado, la consulta HTTP al final del SQS consumer genera incertidumbre, este proceso puede fallar en la conexión porque no se puede acceder al servidor, un error de conexión, un error de servidor, o un código de error identificable.

Finalmente la última fuente de fallas en el procesamiento de mensajes, particularmente en la lambda SQS Consumer es Timeout, una función lambda tiene una duración máxima de dos minutos. Esto limita el procesamiento en grupo de mensajes por la lambda SQS Consumer y puede causar que la lambda genere una ejecución fallida. Sea cual sea el error, el prototipo Chaski solo identifica el problema, no hay sistema de recuperación o respuesta.

3.2.5. Casos reales

La necesidad de manejar adecuadamente errores en la entrega de mensajes se hizo urgente para Adereso de cara a contratos de empresas más exigentes, en los cuales se compromete una calidad de servicio mínima y se dan herramientas para forzar este desempeño por parte de los clientes. A continuación se describirán situaciones que se desean controlar con el nuevo sistema de manejo de intentos y errores.

Uno de los servicios que consumen las instancias de Adereso Helpdesk es Botcenter, un servidor de bots que se encarga de monitorear, almacenar y ejecutar interacciones con bots diseñados e implementados por Adereso, o integrados de otras empresas proveedoras de bots. Este servicio es monolítico y está montado en una máquina virtual vendida por Google Cloud. El problema del servicio es que no es escalable y tiene 3 puntos de falla observados: sobrecarga del servidor de API, sobrecarga del servidor MySQL y sobrecarga del servidor de Cassandra. Sea cual sea la razón de falla el efecto es que conversaciones con los bots se congelan y finalmente se pierden las interacciones. En este caso no hay pérdida de datos, los mensajes se crean en el helpdesk y la respuesta del bot es la que falla, pero por lo general

estas sobrecargas se solucionan solas, en caso de que fueran eventos aislados de demanda, o al poco tiempo de ser detectadas, en caso de que se aumentaran los recursos manualmente para responder al evento. Un sistema de reintentos como el que se desea implementar permitiría a la aplicación recuperarse automáticamente una vez se haya recuperado Botcenter de sus problemas.

Otro tipo de problema que se ha detectado y se espera solucionar se produce debido a falla de los servidores que ofrecen interfaces a los proveedores de mensajes. No todos los mensajes de proveedores de canales de comunicación están acoplados a Adereso Helpdesk, algunos, como el canal de conexión a twitter, son manejados por un microservicio que externaliza el endpoint para que los proveedores envíen sus eventos de mensajes. Esto produce dos potenciales pérdidas de mensajes. Por ejemplo cuando desde podcenter se quiere realizar una acción no esencial para el procesamiento de mensajes en el servicio externo. Ejemplo, cuando se recibe un mensaje desde facebook Adereso Helpdesk hace una consulta pidiendo los datos del usuario autor de este mensaje, esta consulta puede fallar haciendo que se pierda el mensaje. La consulta para pedir datos de usuario puede fallar por ejemplo porque facebook detectó que Adereso está haciendo demasiadas consultas, el endpoint se bloquea temporalmente y luego automáticamente se desbloquea. Con el nuevo sistema de reintentos el mensaje podría fallar en su procesamiento en primera instancia, quizás un par de veces, pero eventualmente el servicio se restaura automáticamente, y en esa ocasión el mensaje se podrá procesar correctamente. Si la falla es permanente, por ejemplo el usuario cambió sus credenciales pero no actualizó su conexión en Adereso Helpdesk, el mensaje fallara siempre, pero se podrá almacenar para que una vez el cliente repare su cuenta, los mensajes guardados puedan ser procesados.

3.3. Características faltantes

En la definición del proyecto no se desea cambiar la funcionalidad del servicio, en su calidad de repartidor de mensajes el prototipo Chaski cumple con lo solicitado por Adereso. Sin embargo, en términos de calidad de software y confianza en el producto, el prototipo se encontraba incompleto.

El proyecto Chaski 2.0 está definido bajo la premisa de que el prototipo desarrollado por el alumno durante su práctica, Chaski 1.0, satisface los requerimientos funcionales que la empresa necesita: existe una APi disponible para enviar mensajes desde un cliente hasta un servidor, los mensajes se entregan en orden, y el servicio es completamente serverless.

Recordemos que Adereso Helpdesk es un servicio que busca facilitar a sus clientes el manejar grandes cantidades de conversaciones provenientes de diversos canales de comunicación. Si por algún motivo los servicios de Adereso se caen, los canales de comunicación seguirán abiertos y se seguirán generando conversaciones.

3.3.1. Asegurar la persistencia de la información

El requisito más crítico del proyecto responde a la necesidad de la empresa de asegurar a sus clientes que los datos de sus conversaciones no se perderán, el servicio puede fallar pero cuando se retome la funcionalidad todos los datos generados durante la falla se deben poder recuperar. El prototipo Chaski no implementa ninguna funcionalidad de almacenamiento de mensajes que no pudieran ser enviados durante su procesamiento, se entiende que si un mensaje fue entregado exitosamente, las responsabilidades de Chaski con los datos terminan.

Adereso necesita que Chaski implemente un sistema para almacenar los mensajes que no pudieran ser enviados con éxito, y poder acceder a estos datos. En particular una base de datos.

3.3.2. Robustez del servicio

Se plantean dos necesidades de robustez del servicio. Primero, implementar un software que toma control de sus potenciales errores y responde a ellos evitando salidas inesperadas del flujo de datos. Segundo, en caso de errores de envío de mensajes, implementar un sistema que permita reintentar el despacho del mensaje, sin saturar la plataforma a la que estas consultas deben llegar.

Una salida errada del programa pone en peligro la integridad del servicio y la de los datos contenidos en el mensaje siendo procesado. Más aún, un servicio como Chaski maneja muchos mensajes, todos estos datos corren peligro.

3.3.3. Logs y diagnóstico de problemas

No se puede esperar que la implementación de Chaski sea un software perfecto, por eso es importante acompañar el desarrollo del servicio con un patrón de reportes útil y conciso. Se necesita un servicio de registros que permita encontrar errores de ejecución, poder diagnosticar la causa de estos errores con seguridad y que ayude a diseñar una solución a la situación.

3.3.4. Monitoreo de funcionamiento

Acompañado con las mejoras ya descritas se necesita que el servicio de monitoreo entregue datos más detallados sobre el funcionamiento de la plataforma. El principal objetivo de un buen sistema de monitoreo es poder revisar el estado de salud del servicio mientras aun funciona y adelantarse a fallas por carga o mal uso del sistema.

3.4. Relevancia

Chaski 1.0 en su periodo de pruebas ha demostrado ser una buena idea, sin embargo se ha usado en paralelo con el sistema de procesos pesados consumer y producer que el proyecto busca reemplazar. Chaski 1.0 no estaba siendo ocupado como en el principal repartidor de mensajes pues la empresa no confía que sea lo suficientemente robusto para cumplir con las expectativas de un software de uso en producción para una empresa cuyos clientes esperan un mínimo de calidad del servicio.

Ya se ha discutido el costo monetario y el esfuerzo humano que conlleva el mantener un servidor para un servicio de broker de mensajes. Es de suma importancia para la empresa el poder deprecar estos servicios y no se puede hacer hasta tener seguridad de que Chaski no perderá mensajes que buscaban ser entregados.

Si el proyecto de tesis tiene éxito en entregar un software de calidad y que de confianza a la empresa de que sus datos no se perderán, el servicio Chaski se podría convertir en un eslabón fuerte de la cadena de producción de Adereso. El desarrollo de nuevas características de Adereso Helpdesk podrá realizarse con la seguridad de que el repartidor de mensajes no presentará problemas inesperados. Esto es de suma importancia en una empresa que busca rediseñar su aplicación monolítica en una con microservicios, la comunicación entre los servicios diseñados debe ser perfecta o el esquema completo se empieza a caer.

En lo que a comunicación interna entre servicios de Adereso refiere, es muy importante destacar que Chaski no es el único medio de comunicación entre potenciales microservicios, Pub/Sub un servicio comunicación de la red de productos de Google que soluciona el problema de Broadcast/Subscriber también es muy importante en la arquitectura de microservicios de la empresa, pero Chaski resuelve otro problema que no se puede atacar con pubsub directamente, y en general el objetivo es desarrollar o implementar las herramientas correctas y eficientes para cada problema encontrado en el proceso de desarrollo.

3.5. Objetivos

Mejorar el prototipo Chaski, sistema repartidor de mensajes desarrollado por el autor, para poder ser usado en plenitud por la empresa Adereso.

3.5.1. Objetivos específicos

1. Asegurar la trazabilidad del 99 % de los mensajes enviados por Adereso Helpdesk al servicio Chaski
2. Asegurar que en caso de fallas que produzcan pérdidas de mensajes se mantenga registro del 90 % de los datos.
3. Desarrollar sistema de reintentos de envío de mensajes.
4. Mejorar el sistema de monitoreo, considerando métricas de origen y destino.

5. Mejorar el sistema de logs para facilitar la detección de problemas.

3.5.2. Deseables

1. Reducir el tiempo de ejecución de las lambdas.
2. Desarrollar sistema de alarmas para poder detectar problemas en el funcionamiento del servicio.

Capítulo 4

Solución

4.1. Arquitectura del servicio

El servicio Chaski fue concebido como un microservicio que hace uso de productos en la nube en la mayor medida posible para facilitar el desarrollo. Esta decisión busca lograr que Chaski sea un software fácilmente escalable y de alta disponibilidad.

El servicio Chaski se incorpora en el proceso productivo de Adereso como un intermediario entre distintos elementos de la aplicación, en particular entre productores de mensajes (servicios que exponen un endpoint para que proveedores de canales de comunicación envíen eventos de mensajes) y consumidores (instancias de ejecución de Adereso Helpdesk), y de regreso.

Los productores de mensajes se comunican por medio de una interfaz HTTP con el servicio Chaski. La respuesta de la invocación de Chaski es referente solo a la recepción del evento, el procesamiento del mensaje en Chaski es asíncrono. En el otro extremo, Chaski llama un endpoint HTTP para ejecutar la acción incluida en el mensaje procesado.

Debido a que Chaski se construye plenamente con servicios web ya existentes, toda la arquitectura de hardware es transparente al desarrollador, no hay control sobre las máquinas en las que se instalan los servicios utilizados. Con una excepción, los reportes de Chaski se envían a un servidor ELK manejado por Adereso. Desde el punto de vista de Chaki este es otro servicio disponible por internet, sin embargo, la arquitectura de este servicio es conocida y manejada por la empresa.

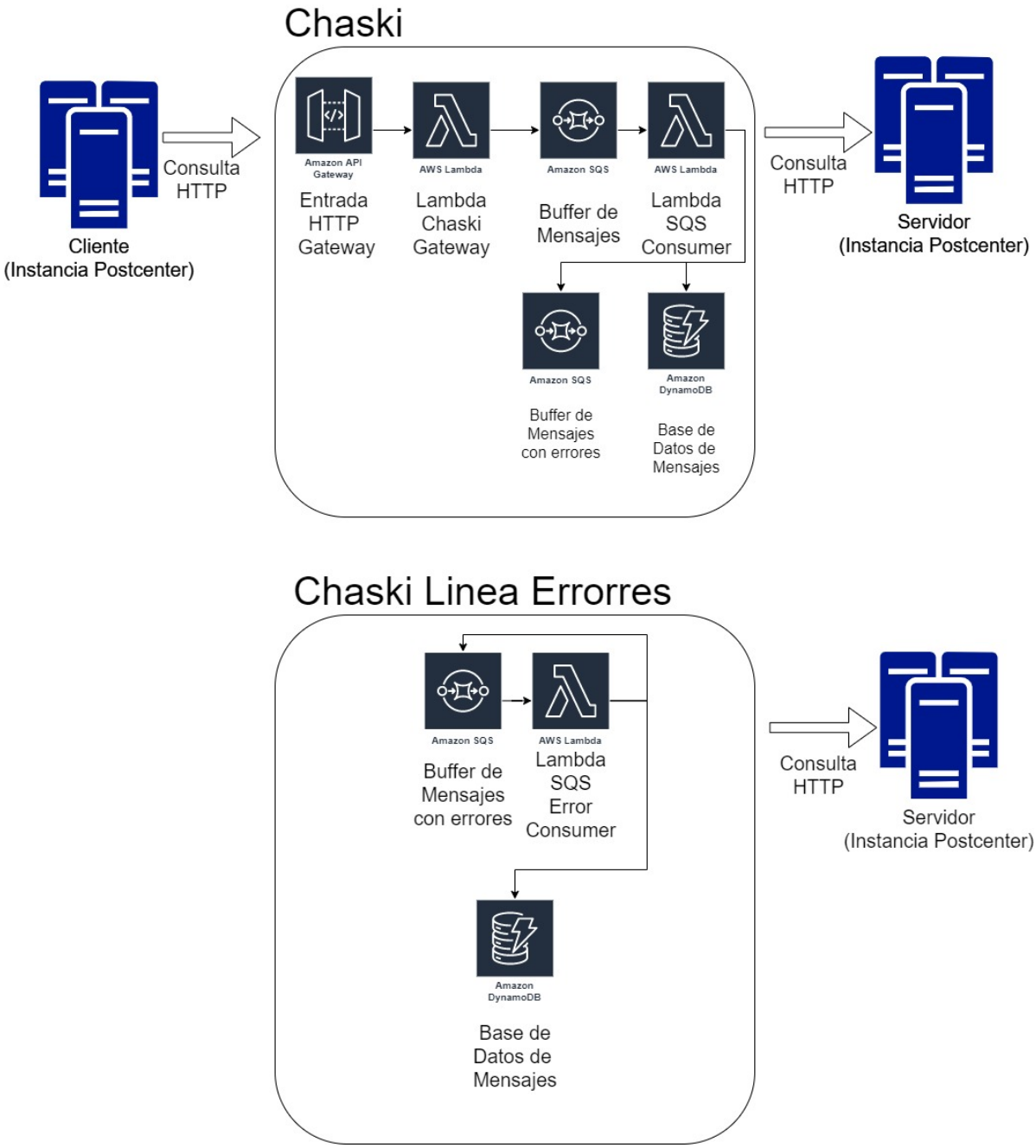


Figura 4.1: Esquema general de diseño del servicio Chaski desarrollado.

4.2. Estructura de la solución

El software entregado sigue la estructura básica del prototipo, se usan servicios en la nube para evitar tener que volver a implementar soluciones que están disponibles en el mercado, y se usó AWS Lambda para conectar estos productos y dar lógica a la solución.

Un desafío importante de la solución es la duplicación de código. Se buscó la cantidad de líneas repetidas entre las lambdas encapsulando funcionalidad en librerías. Estas librerías también buscan facilitar futuras modificaciones al servicio.

En esencia el nuevo servicio Chaski se conforma de una API de entrada implementada con API Gateway, una línea de procesamiento inicial que busca enviar el mensaje a su destino, y una línea de procesamiento paralela diseñada para reintentar el envío de mensajes que fallaron en el camino principal. Las funciones lambda se comunican guardando mensajes en colas SQS. Finalmente, los mensajes completamente fallidos se guardan en una base de datos DynamoDB.

Los servicios anexos usados en el funcionamiento de Chaski son, Datadog como servicio de monitoreo y ELK como servicio de reportes y de métricas personalizadas. Como servicio de CI se sigue usando DroneCI y los paquetes de despliegue de Chaski se guardan en Amazon S3.

4.2.1. Servicios usados

En esta sección se detallarán las responsabilidades de los servicios usados en la construcción de Chaski.

- Amazon API Gateway: se encarga de defnir la interfaz HTTP, los endpoints y la seguridad de al conexion. Se definieron 3 endpoint, el de produccion, el de canary y el de desarrollo.
- Amazon Lambda: se encarga de levantar los bloques logicos desarrollados. Se usa para conectar todos los elementos de la solucion, API Gateway, SQS y DynamoDB.
- Amazon SQS: se encarga de almacenar mensajes, el buffer de mensajes. Se usa en formato FIFO para asegurar el orden. Se desplegaron varias colas SQS con el objetivo de permitir procesamiento paralelo de mensajes.
- DynamoDB: es la base de datos usada para almacenar los mensajes que no pudieron ser entregados exitosamente.
- ELK: el servicio ELK es un conjunto de servicios necesarios para permitir el monitoreo, acceso a registros y métricas.

4.2.2. Librerías escritas

Message Wrapper

Todas las lambdas en el proyecto reciben los mensajes como un string desde el punto anterior de procesamiento, las colas SQS que hacen de buffer. El Message Wrapper se encarga de encapsular la funcionalidad de parsear el string en un objeto de mensaje y de construir el string que debe ser almacenado en la cola objetivo de la lambda. Esta librería define y lanza excepciones de mensajes incompletos o mensajes mal parseados al momento de imprimir el string de salida o de parsear el objeto de mensaje respectivamente.

Al hacer una librería con esta responsabilidad nos aseguramos de que todas las lambdas esperen el mismo tipo de mensaje y entreguen un mensaje válido. Además, esto ayuda a reducir la cantidad de código duplicado entre las distintas lambdas, en particular SQS Consumer y SQS Error Consumer comparten parte importante de su código.

Chaski Requests

El objetivo de la librería Chaski request es encapsular la construcción y realización de una consulta HTTP, y evaluar el resultado recibido. Esta librería permite reducir duplicación de código relacionada con el manejo de la respuesta HTTP entre las lambdas Consumer, que son las que realizan la request al servicio externo.

La lambda llama a la librería con los parámetros relevantes y Chaski request encapsula los errores de ejecución, o los errores por código de estado de la respuesta y levanta una excepción. De esta forma cada lambda de Consumer puede responder a la excepción como corresponde, pero no evalúa la respuesta para determinar si hay o no un error.

Dynamo Handler

Esta librería maneja las credenciales y es una abstracción de la librería estándar de escritura en Dynamodb que recibe los datos desde las lambdas de Chaski y genera el diccionario que será insertado en la base de datos. La importancia de esta función es abstraer el diccionario que se inserta en DynamoDB en parámetros de ejecución del método.

Una característica de la librería es que permite el procesamiento en batch de mensajes a DynamoDB, no se usa esta funcionalidad, pero existe.

Chaski Logger

Chaski logger es un wrapper de la librería standard de logging de python, el objetivo de la librería es dar una interfaz que permita dar formato json a los mensajes de registro con el fin de estandarizar la función de procesamiento de mensajes de Logstash en el núcleo ELK de Adereso.

Para hacer funcionar el servicio de logs para Chaski además se debió configurar un input en el logstash del servicio ELK de Adereso, esté input recibe el mensaje formateado desde Amazon ClodWatch. El mensaje es procesado por un filtro Grok que usando una expresión regular extrae la información de los datos recibidos.

Finalmente para enviar los datos desde Amazon Cloudwatch al servicio ELK de Adereso se usó una lambda desarrollada por Elastic llamada function beat, que lee los stream de cloudwatch y realiza la consulta HTTP al input de logstash configurado en el ELK de la empresa.

Se definieron dos interfaces de uso para la librería Chaski Logger, una para enviar men-

sajes de logs, para monitorear el funcionamiento y debuggear. Otra para enviar mensajes de métricas personalizadas, el formato está pensado para facilitar la definición de visualizaciones y búsqueda en Kibana.

4.2.3. Descripción de las lambdas

API Gateway

Api gateway es la lambda que recibe eventos del servicio de Amazon con el mismo nombre. Las responsabilidades de esta lambda son determinar si la consulta recibida fue a la ruta correcta, y si el contenido del mensaje es válido, y determinar la cola a la que debe llegar el mensaje, esto último depende del subject incluido en el mensaje. Los posibles destinos de un mensaje son, colas de mensajes y la cola de asignaciones. Ambas colas son equivalentes desde el punto de vista de chaski, pero se usan por separado porque en producción los mensajes en estas colas cumplen objetivos distintos.

SQS Consumer

SQS es la lambda que se encarga de manejar los eventos de salida de las diferentes colas de mensajes de Chaski.

El primer paso del procesamiento de la lambda SQS Consumer es revisar que el mensaje está correctamente formateado, de esto se encarga la librería message wrapper. Luego se intenta mandar la consulta HTTP al servidor remoto. Para eso solo se invoca la librería chaski request con los parámetros, ya se sabe que los parámetros existen pues se pasó la revisión del mensaje.

La consulta puede resultar en un envío exitoso o diversos errores, si el mensaje se envió exitosamente el procesamiento del mensaje terminó. De haber fallado la lambda SQS Consumer se encarga de empaquetar el tipo de error en el mensaje y ponerlo en la cola de entrada para el procesamiento de mensajes de errores.

SQS Error Consumer

SQS Error Consumer se encarga de procesar los mensaje de eventos generados por colas de errores, cumple la misma función de SQS Consumer, intentar mandar el mensaje a través de una consulta HTTP, pero además debe revisar el tipo de error, y la cantidad de veces que se ha intentado enviar el mensaje para determinar qué se debe hacer, si volver a intentar o guardar el mensaje en la base de datos.

El primer paso del procesamiento de la lambda SQS Error Consumer es revisar que el mensaje esté correctamente construido, de esto se encarga la librería message wrapper. Luego la lambda debe revisar el error que envió el mensaje a la cola de errores y la cantidad de veces que se ha reintentado el envío del mensaje, con estos datos se determina el próximo paso. Una

de las opciones que se debe intentar enviar el mensaje de nuevo, de tener éxito en el envío el procesamiento del mensaje terminó, de fallar se envía a la cola de errores correspondiente. Por otro lado, si ya se alcanzó el número máximo de reintentos definidos para el error encontrado la primera vez que se intentó realizar la consulta el mensaje se guarda en la base de datos de mensajes.

El protocolo de reintentos se define en un diccionario para cada tipo de error que se espera que chaski maneje, estos son, errores de conexión, errores 400 y errores 500, no se entra en detalles más específicos, pero se podría hacer un desarrollo para por ejemplo tener un protocolo diferente para los errores código 404 y para 403. El diccionario para cada tipo de error debe incluir un valor "reintentos por cola" y una lista de nombres de colas sqs. La lambda revisa este diccionario para el error del mensaje, la cantidad máxima de intentos es cantidad de reintentos por cola multiplicado por el largo de la lista de colas. La cola a la que se debe enviar el mensaje si aún hay intentos disponibles se calcula calculando el cociente entre reintentos actuales y reintentos por cola.

4.3. Sistema de reportes

El sistema de reporte se instaló en una plataforma ELK administrada por Adereso, el punto de entrada de los nuevos mensajes de registro de actividad es un servidor logstash que procesa las consultas y guarda los datos en una base de datos elasticsearch. Todos los reportes y métricas derivadas de los datos enviados se pueden acceder por el servidor de Kibana, el front de este sistema.

Para enviar los mensajes de reportes desde chaski, se usó una lambda function beat, entregada por elastic que fue configurada para poder alimentar logstash. Los mensajes enviados a logstash, a través de function beat fueron formateados a representación json; este formateo es responsabilidad de la librería chaski logger.

El mensaje que se envía a logstash es el que se imprime a cloudwatch, este incluye tipo de mensaje, un timestamp, un identificador de ejecución y el mensaje que se desea enviar. Por este motivo uno de los filtros aplicados al mensaje que se recibe en logstash es un grok. En términos simples se hace un match del mensaje recibido con una expresión regular, este filtro permite guardar distintos segmentos del texto que hicieron match con las diferentes partes de la expresión regular. De esta forma se extrae el tipo de mensaje de reporte, el timestamp entregado por cloudwatch, el identificador de ejecución y el mensaje en cuestión. Además como el mensaje fue formateado a json, este json también se puede interpretar por un filtro que intenta generar el diccionario, esto permite guardar el objeto json en la base de datos y hacer consultas por las llaves del json.

Finalmente, el último desarrollo referente a reportes fue diseñar un dashboard que muestra las métricas generadas, y los mensajes de logs de forma ordenada. Es importante mencionar que Elasticsearch entrega herramientas de búsqueda de texto bastante poderosas, esto permite revisar eficientemente los reportes y generar métricas a partir de mensajes correctamente formateados en json.

Base de datos de mensajes

Los mensajes que no pudieron ser enviados con éxito en ninguno de sus intentos son almacenados en una base de datos Dynamodb. La llave primaria de la tabla de mensajes es la fecha "día-mes-año" en que se guardó el mensaje, la de ordenamiento es el timestamp.

La base de datos Dynamodb es una no relacional que permite guardar datos en formato json como campos de la fila en la tabla. Gracias a esto podemos guardar toda la información del mensaje en la tabla. Además, se guarda en un campo el tipo de error que envió el mensaje a la cola de manejo de errores.

Esta estructura busca poder determinar rápidamente todos los errores que ocurrieron en cierto delta de tiempo, desde la interfaz de Dynamodb, y luego en memoria filtrar y organizar estos datos por contenido del mensaje o tipo de error.

4.4. Sistema de Reintentos

El sistema de reintento funciona volviendo a encolar mensajes en la cola de errores correspondiente. La decisión de si encolar o no y en qué cola hacerlo, depende del procesamiento del mensaje y de un diccionario de configuración.

El diccionario debe contener un lista de colas destino y una cantidad de reintentos por cola para cada error definido en el proyecto; de conexión, de timeout, código HTTP 400 y código 500. Este sistema permite cambiar el protocolo de reintentos de acuerdo a las necesidades de la empresa de una forma simple. Si bien se definieron 3 colas, Fast, Medium y Slow, estas distinciones solo tienen significado semántico, el tiempo de retención del mensaje depende exclusivamente de la configuración de la cola y la función SQS Error Consumer no tiene control alguno sobre esto. Se asume que las referencias están bien configuradas cuando se despliega Chaski.

4.5. Ciclo de vida de un mensaje

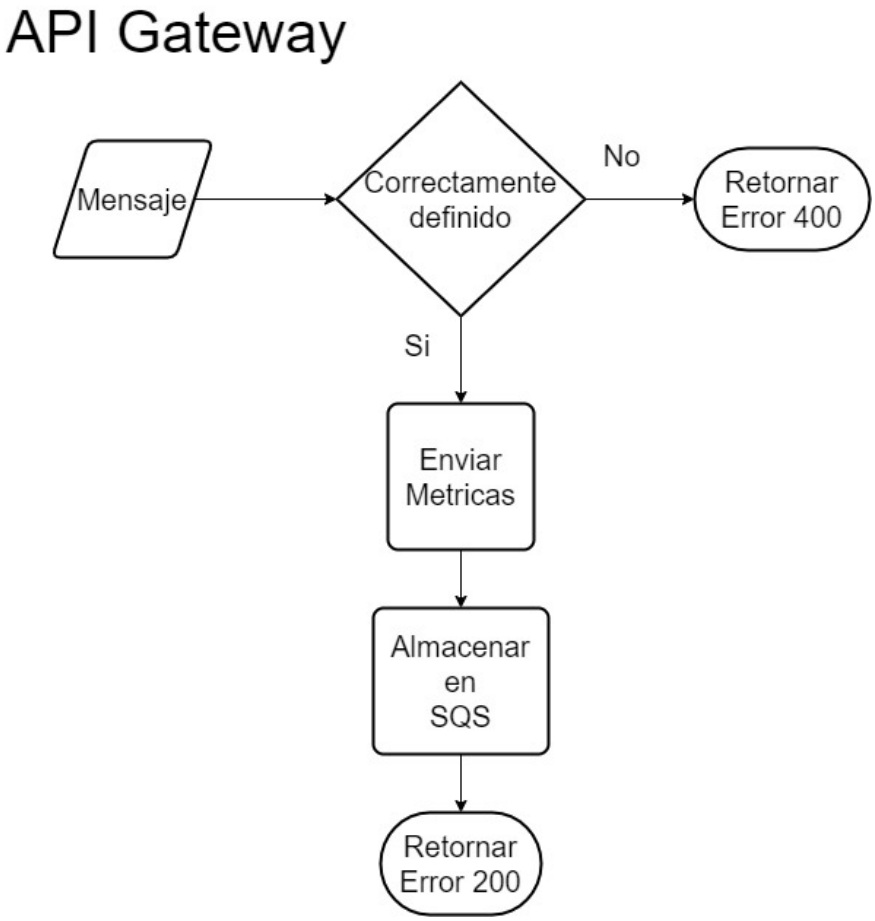


Figura 4.2: Esquema ciclo de vida de un mensaje en la lambda API Gateway.

Un mensaje es recibido por el servicio API Gateway de Amazon, este levanta un evento para la lambda Api Gateway. Esta llamada puede responder con error 400, si el mensaje no viene con todos los parámetros necesarios, error que es comunicado al cliente que invocó Chaski, o continúa con el procesamiento del mensaje encolando el mensaje en la cola correspondiente determinada a partir del subject del mensaje. Si la función lambda falla por motivos inesperados, API Gateway automáticamente envía un error 500 al cliente.

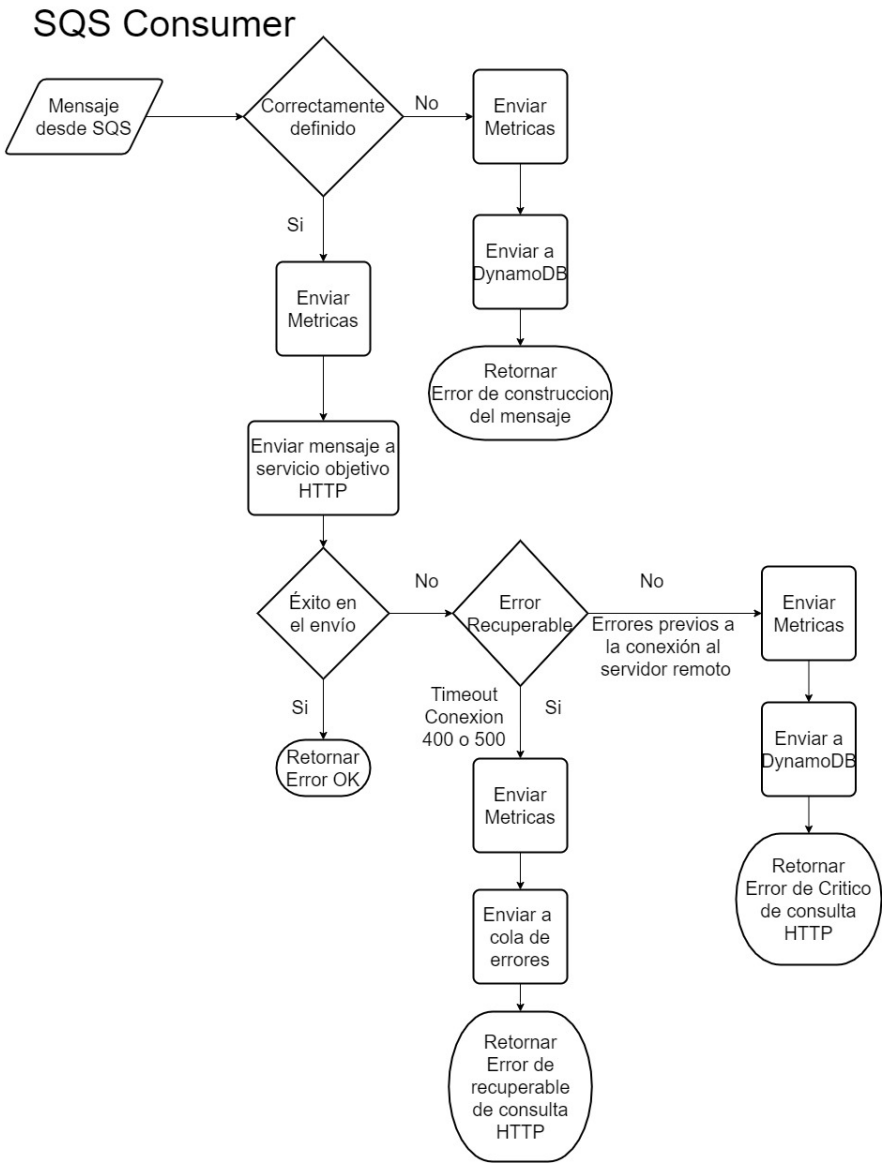


Figura 4.3: Esquema ciclo de vida de un mensaje en la lambda SQS Consumer.

Las colas SQS de mensajes levantan eventos automáticamente los cuales son capturados por la lambda SQS Consumer. Esta lambda revisa la correctitud del evento recibido e intenta enviar un mensaje, si este mensaje no pudo ser enviado con éxito entonces es encolado en la cola de errores indicando en su estructura el motivo de la falla. Si la consulta al servidor objetivo es exitosa el mensaje termina su procesamiento en Chaski.

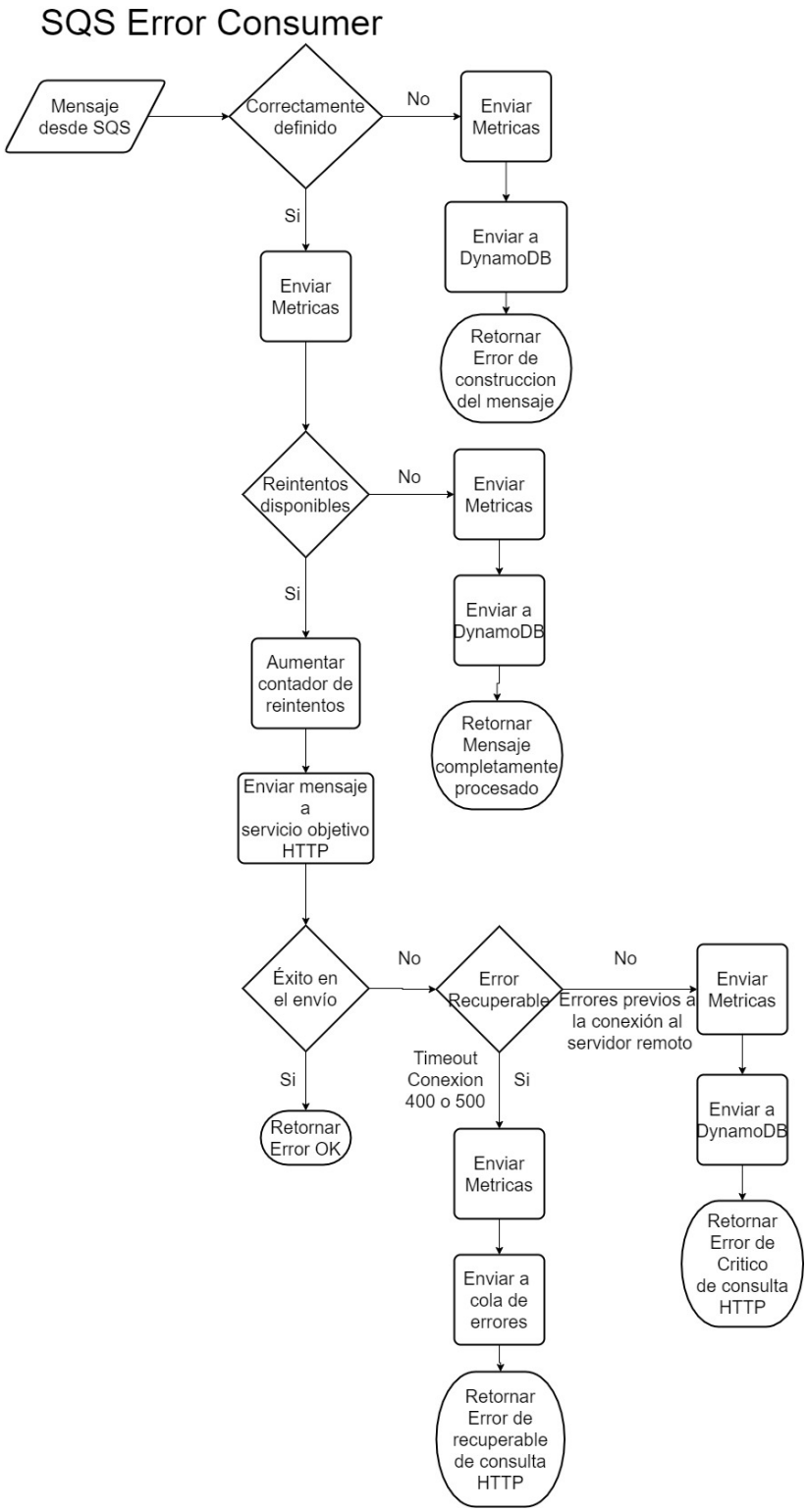


Figura 4.4: Esquema ciclo de vida de un mensaje en la lambda SQS Error Consumer.

Finalmente la cola errores se encarga de levantar eventos de mensajes para la lambda SQS Error Consumer. Esta lambda revisa la correctitud del mensaje, luego determina el siguiente paso, si el mensaje agotó su numero de reintentos se guarda en la base de datos, si el mensaje puede seguir siendo procesado se realiza la consulta HTTP al servidor remoto, de fallar se

encola nuevamente y de ser exitoso se terminó el procesamiento del mensaje en Chaski.

4.6. Despliegue de la Solucion

4.6.1. Deployment fallido

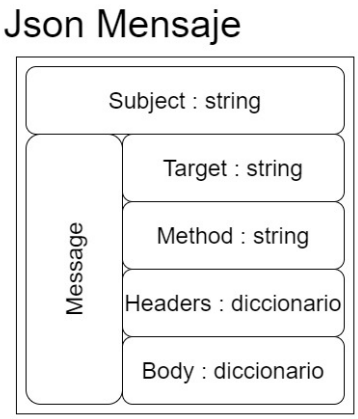


Figura 4.5: Json esperado por el prototipo Chaski

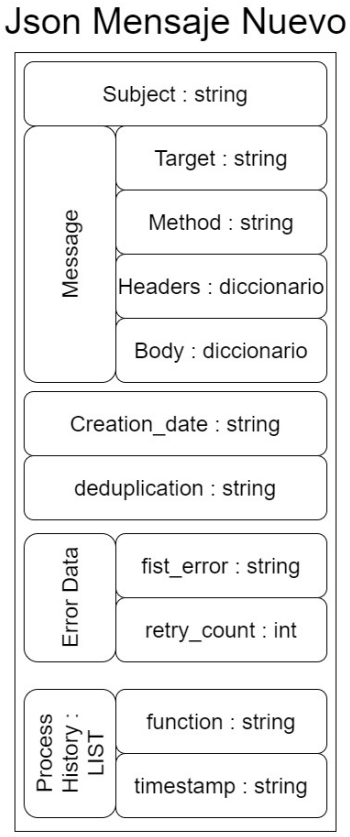


Figura 4.6: Json esperado por el nuevo Chaski, fallido

En su primer despliegue el código desarrollado no pudo ser desplegado exitosamente en la rama de producción de Chaski usada por Adereso. El despliegue se intentó hacer usando la funcionalidad de versiones y aliases de las lambda de AWS para balancear la carga, con el objetivo de hacer que solo el 10 % del tráfico sea procesado por la nueva versión de Chaski. Este procedimiento no fue efectivo al intentar separar las lambdas pues una vez ingresado un mensaje en la cola SQS no se puede asegurar que el 10 % de mensajes que serían manejados por la lambda SQS Consumer nueva hayan sido procesados por la lambda API Gateway nueva también. Esta inconsistencia en el procesamiento se produjo por incompatibilidad entre el objeto json, descrito en la figura 4.6, encolado en SQS por la lambda API Gateway original y el formato de mensaje esperado por la lambda SQS Consumer nueva, descrito en la figura 4.7. Todos los mensajes que se procesaron por el nuevo Chaski se perdieron.

Debido a este error de funcionamiento se revirtió el cambio. Para intentar salvar la mayor cantidad de trabajo posible, se planificó implementar y desplegar una nueva versión Chaski 2.0. En este nuevo proceso de desarrollo se pretende solucionar dos errores críticos de planificación, el primero de estos errores fue trabajar el proyecto pensando en una única gran entrega, el segundo fue no considerar el impacto de los cambios realizados en los datos ya procesados en el momento de desplegar la nueva solución.

De este deploy se pudo rescatar la conexión de registros de actividad con el servicio ELK de Adereso. Esta funcionalidad es externa al código de las lambdas, por eso se pudo entregar por separado. Sin embargo, el formato de los mensajes no es el esperado para definir las visualizaciones y dashboards.

4.6.2. Solución implementada: Asegurar retrocompatibilidad de las lambdas

No hay ningún motivo de fondo para haber desarrollado un código no retrocompatible, las lambdas y librerías se hicieron de esta forma por falta de visión al momento de planificar el proyecto. La solución propuesta es bastante simple, el json original de mensaje contiene toda la información necesaria para procesar un mensaje en Chaski. Todos los otros parámetros son opcionales. En este sentido no se asume en ninguna lambda cual es la versión de las lambdas que han procesado el mensaje anteriormente.

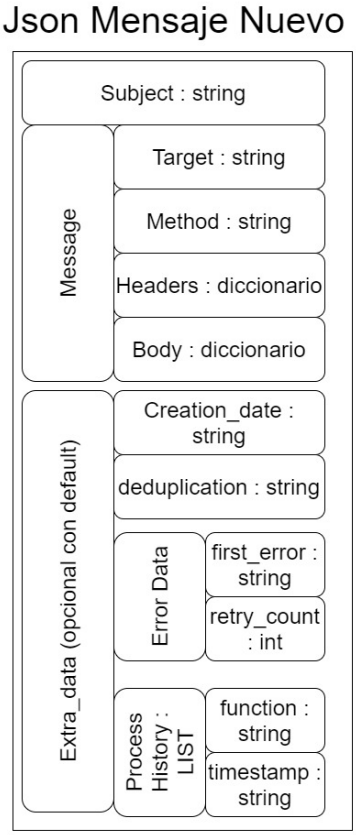


Figura 4.7: Json esperado por el nuevo chaski, retrocompatible

El único cambio profundo del procesamiento de mensajes efectuado es que en la lambda SQS Error Consumer se procesa el mensaje bajo el supuesto de que si el mensaje no viene con metadata extra, entonces no tiene reintentos disponibles y se guarda en DynamoDB sin reintentar su envío. En la figura 4.7 se describe la nueva estructura json que se transmite de una lambda a otra a través de Amazon SQS. En esta estructura todos los parámetros que no pertenecen al json esperado por las lambdas del prototipo Chaski, figura 4.6, se consideran opcionales.

Gracias a la definición de la librería Chaski Wrapper se necesitó hacer el cambio en el manejo de errores de atributos de mensajes en un solo bloque de código. La otra mitad de trabajo para asegurar la retrocompatibilidad de las nuevas lambdas SQS Consumer y SQS Error Consumer es definir un manejo distinto si el mensaje viene con metadatos extra o no. Esto consiste en que si el mensaje no viene con los metadatos el procesamiento es más básico y no se hará uso del nuevo código desarrollado pero se inicializan los metadatos, por otro lado, si los metadatos están correctamente definidos en el mensaje se hará uso de la nueva funcionalidad.

Capítulo 5

Validación

El proceso de validación del trabajo realizado se explicará en tres secciones. La primera se refiere al proceso de desarrollo en Adereso, la segunda a pruebas realizadas en el ambiente QA y la tercera trata los beneficios que ya se obtuvieron en la empresa gracias al desarrollo de Chaski 2.0.

5.1. Proceso de Desarrollo

En Adereso existen tres procesos que se deben completar para poder declarar un desarrollo listo para ser desplegado en producción: evaluación de pares, demostrar coverage mayor a 80 % de las líneas de código que se realiza de forma automática en algún protocolo de integración continua, y proceso de QA en ambientes de prueba similares a los de producción. Estos ambientes de QA son instancias completas del Adereso Helpdesk pero cuyos datos están separados de los de producción.

5.1.1. Sobre el proceso de evaluación de pares

Como el proyecto de desarrollo de Chaski 2.0 fue presentado como propuesta de tesis, toda la responsabilidad fue entregada al alumno. Esto es resulto ser un inconveniente y desafió en el proceso de evaluación pues en general la evaluación de pares funciona porque, los proyectos son realizados en equipo, o los cambios propuestos son pequeños y acotados. Las mejoras a Chaski fueron un gran proyecto que introdujo varios cambios, y fue realizado únicamente por el alumno.

Para soslayar esta situación se optó por hacer una presentación al equipo de desarrollo donde se explicaba como funciona Chaski, qué cambios se hicieron al proyecto y cómo evaluar dichos cambios. Se considera importante mencionar que de haber hecho entregas más pequeñas y recurrentes el proceso de evaluación se habría hecho más fácil, aunque probablemente la reunión explicativa habría sido necesaria de todas formas.

5.1.2. Sobre el proceso de QA

Para realizar QA a Chaski 2.0 se montó toda una infraestructura paralela, sus colas SQS, sus endpoint en API Gateway y sus lambdas necesarias. Esta nueva infraestructura se asoció, por configuración interna, como el servicio usado por todas las instancias de QA de Adereso. Instancias de ejecución de Adereso Helpdesk y todos sus servicios que se usan para hacer control de calidad dentro de la empresa. Con esto se busca evaluar si los cambios realizados afectarán el servicio de formas inesperadas, y si la nueva funcionalidad efectivamente funciona. El proceso de control de calidad es muy bueno para rechazar nuevos desarrollos, si un cambio no funciona en QA, definitivamente no funcionará en producción. Pero un desarrollo que pasa exitosamente su proceso de QA puede presentar problemas no esperados en producción, no se puede evaluar cada línea de código y cada caso de uso, pero el proceso busca llegar a ese nivel de efectividad. Por esto el despliegue del nuevo código se hace bajo un protocolo de Canary, liberando la característica poco a poco a la aplicación general.

Las aceptaciones falsas de desarrollos pese a efectuarse un proceso de control de calidad se debe a que los ambientes de QA, que si bien tiene todos los elementos de la aplicación desplegados los tiene a una escala mucho menor que en producción, por temas de costos y de uso. Además, porque la Aplicación Helpdesk es bastante extensiva y evaluar cada interacción es infactible por costos y por eficiencia del proceso.

5.1.3. Cobertura e integración continua

Parte del tiempo de desarrollo de la tesis se dedicó a escribir pruebas unitarias del código de Chaski y de las librerías creadas para el proyecto. Se desarrollaron tests unitarios para las lambdas, API Gateway, SQS Consumer y SQS Error Consumer. También se desarrollaron pruebas para las librerías: Message Wrapper y Chaski Request. Con esto se logró llegar a un 86 % de coverage del proyecto.

El porcentaje de coverage faltante se justifica en ramas de código para excepciones generales, que se usan para atrapar cualquier error dentro del procesamiento de la función, en general no se debería llegar a estas excepciones. Y en las librerías: DynamoHandler, ChaskiLogger y ChaskiHashlibs. Estas librerías no se testean porque en general solo implementan una librería importada, boto3 para DynamoHandler, logger para ChaskiLogger y murmurhash para ChaskiHashlibs, solo se estaban testeando las firmas de las funciones.

5.2. Pruebas realizadas en ambiente QA

Esta sección explicarán unas pruebas de funcionamiento realizadas en el ambiente de QA, las que fueron independientes del control de calidad realizado y buscan evaluar que el funcionamiento sea el esperado y poder cuantificar los resultados obtenidos.

5.2.1. Explicación de la prueba

La prueba consiste en enviar desde un cliente HTTP local una cantidad configurable de mensajes, en este caso doscientos mensajes. Estos mensajes llegan a Chaski, son procesados y el requisito final llega a un servidor implementado para estas pruebas. El servidor está implementado para responder con código HTTP 200, 400, 500 o con un 200 tras un tiempo de espera (este último para producir timeouts en la request desde chaski).

El cliente local que inicia el proceso de prueba construye un mensaje para Chaski aceptable definiendo especialmente la URL objetivo del mensaje. Esta url puede ir al endpoint que retorna 200, 400, 500 o timeout. Se envían cincuenta mensajes a cada uno de estos endpoints (este valor es configurable). Chaski retorna a la consulta HTTP con 200 si el mensaje es aceptado para ser procesado o un error si no, en la prueba todos los mensajes son aceptados. En este punto el cliente puede olvidarse del procesamiento de los mensajes.

El servidor para las pruebas se desarrolló en Python 3.8 y con framework Django. El software tiene un modelo de base de datos para almacenar un historial de las consultas recibidas por el servidor. Y tiene cuatro endpoints, todos retornan un mensaje vacío con un código HTTP específico, dos retornan código 200, pero uno se demora en responder (buscando levantar error de timeout en el cliente que lo llama), uno retorna código 400 y uno retorna código 500.

5.2.2. Resultados de la prueba

En la prueba se enviaron doscientos mensajes a Chaski 2.0, estos mensajes se dividen en cuatro grupos de cincuenta mensajes cuyos endpoint objetivos son cada uno de los cuatro endpoints de los implementados en el servidor. Por el patrón de reintentos que está implementado en Chaski 2.0 de QA se espera que el cliente reciba ochocientos de mensajes totales.

Endpoint	Enviados	Esperados
200	50	50
400	50	250
500	50	300
Timeout	50	200

Tabla 5.1: La Tabla muestra la cantidad de mensajes enviados a Chaski y la cantidad de consultas esperadas en el servidor de pruebas

Endpoint	Enviados	Repeticiones	Totales
200	50	0	50
400	50	200	250
500	50	250	300
Timeout	50	150	200

Tabla 5.2: La Tabla muestra la cantidad de mensajes enviados a Chaski, la cantidad de repeticiones esperadas y la cantidad de llamadas registradas en el servidor de pruebas

```
>>>
>>> RequestHistory.objects.all().count()
800
>>> RequestHistory.get_200_response_count()
50
>>> RequestHistory.get_400_response_count()
250
>>> RequestHistory.get_500_response_count()
300
>>> RequestHistory.get_timeout_response_count()
200
>>>
```

Figura 5.1: Foto con los resultados de consultar la base de datos del servidor de pruebas.

En la tabla 5.1 podemos ver la distribución de mensajes esperados organizados por endpoint. En la tabla 5.2 y figura 5.1 podemos ver los resultados efectivos. Con esta prueba se puede concluir que desde el punto de vista del cliente y servidor de pruebas hay consistencia y éxito. Además, confirma que los mensajes que llegan a ser procesados por el servidor final de acuerdo al patrón de reintentos definido en el servicio Chaski. Demostrando que esta característica de Chaski 2.0 fue correctamente implementada.

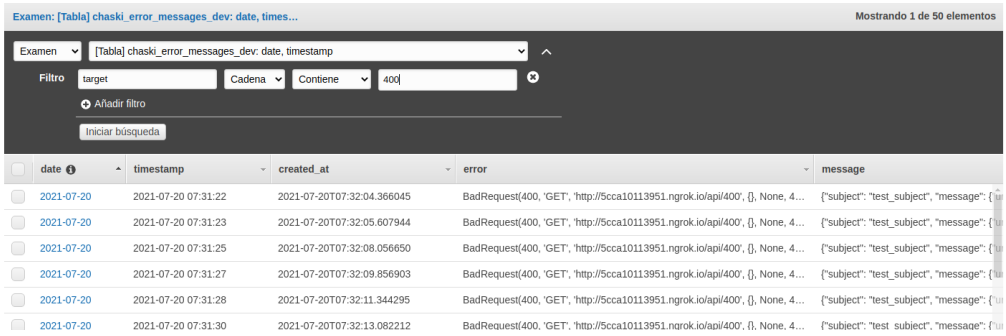


Figura 5.2: Foto con los datos almacenados en DynamoDB cuya url objetivo retorna código HTTP 400.

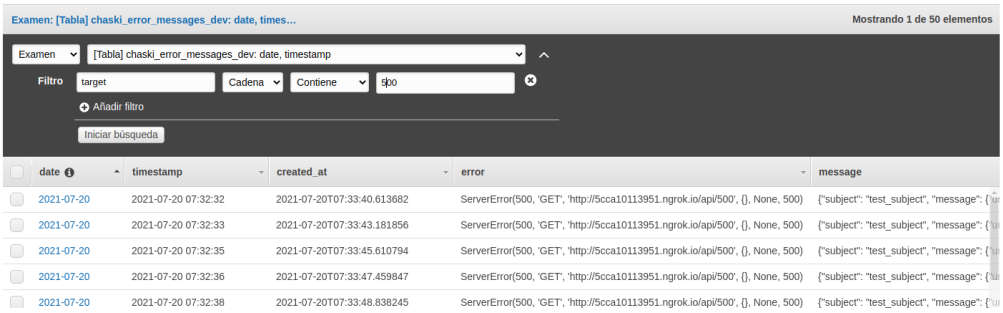


Figura 5.3: Foto con los datos almacenados en DynamoDB cuya url objetivo retorna código HTTP 500.

En las imágenes 5.2, 5.3 y 5.4 podemos ver los registros de los mensajes fallidos en DynamoDB. Esto demuestra que los mensajes que no pudieron ser entregados exitosamente

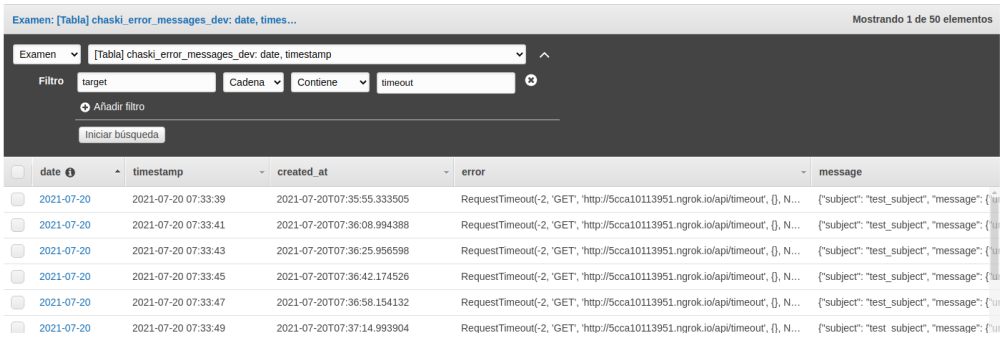


Figura 5.4: Foto con los datos almacenados en DynamoDB cuya url objetivo genera un error de timeout en Chaski.

al servicio objetivo fueron guardados exitosamente, con los datos correctos y cantidades correctas.

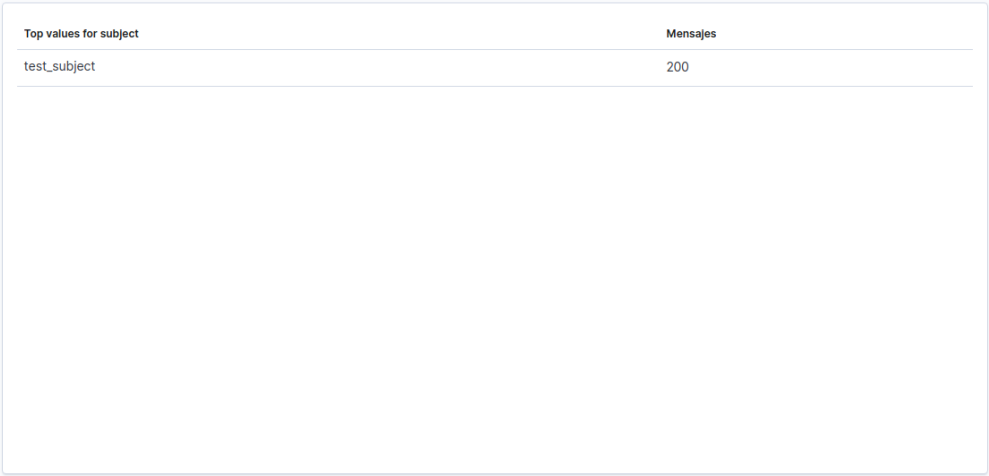


Figura 5.5: Foto con las métricas generadas en la prueba para la métrica messages_by_subject en la lambda SQS Consumer.

En la imagen 5.5 podemos ver el registro de la métrica mensajes por subject que se almacena en Kibana, el subject “test_subject” fue definido en el cliente de la prueba. Esto demuestra que el registro de actividad del servicio Chaski 2.0 también es consistente con los datos que se esperaban obtener.

Top values of json_parsed.error_by_target-target.keyword	Mensajes
http://5cca10113951.ngrok.io/api/500	250
http://5cca10113951.ngrok.io/api/400	243
http://5cca10113951.ngrok.io/api/timeout	150

Figura 5.6: Foto con las métricas generadas en la prueba para la metrica error_by_target en la lambda SQS Error Consumer.

En la imagen 5.6 podemos ver que las métricas de cantidad de errores registrados por subject en la lambda SQS error consumer no fue exitosa. Si bien para los errores 500 y timeout el conteo fue correcto, para el error 400 el contador sobrepasa el valor esperado por cuarenta y tres mensajes. Esto es un error aún no identificado, y se sigue trabajando en mejorar el sistema. Pero en general se declara como éxito parcial, toda vez que el mensaje ingresa en la cola de errores se consideró más crítico que el almacenamiento en DynamoDB funcione correctamente.

5.3. Resultados vistos en producción

Esta sección busca evidenciar de que Chaski 2.0 está funcionando en producción. Se mostrarán algunas imágenes con los distintos puntos donde podemos ver el resultado del trabajo, con lo que se presente mostrar la existencia de los datos y resultados esperados.

5.3.1. Mensajes procesados

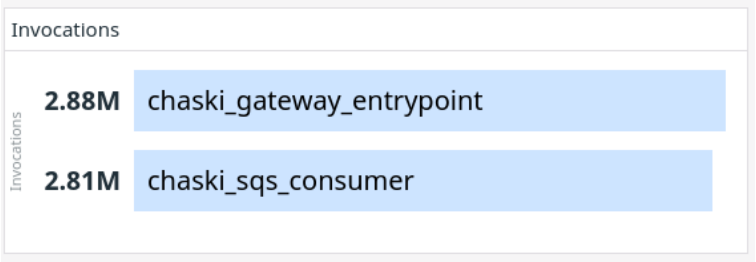


Figura 5.7: Foto con las métricas generadas en Datadog que muestran la cantidad de invocaciones de las lambdas API Gateway y SQS Consumer en Chaski posterior al despliegue de Chaski 2.0.

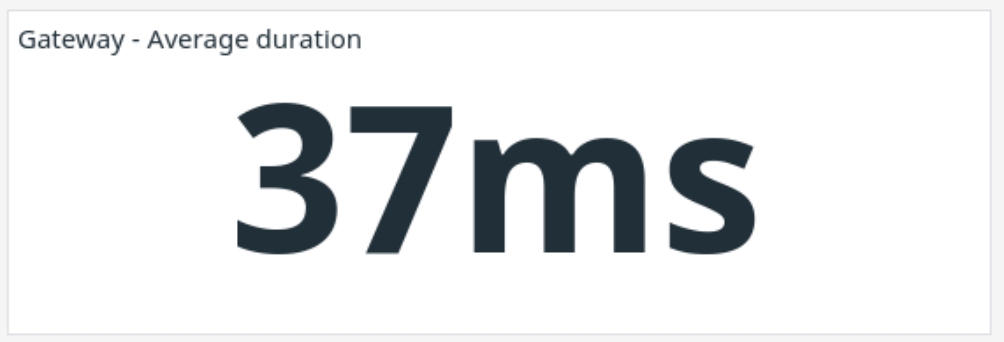


Figura 5.8: Foto con las métricas generadas generadas en Datadog para el tiempo de ejecución de la lambda API Gateway posterior al despliegue de Chaski 2.0.

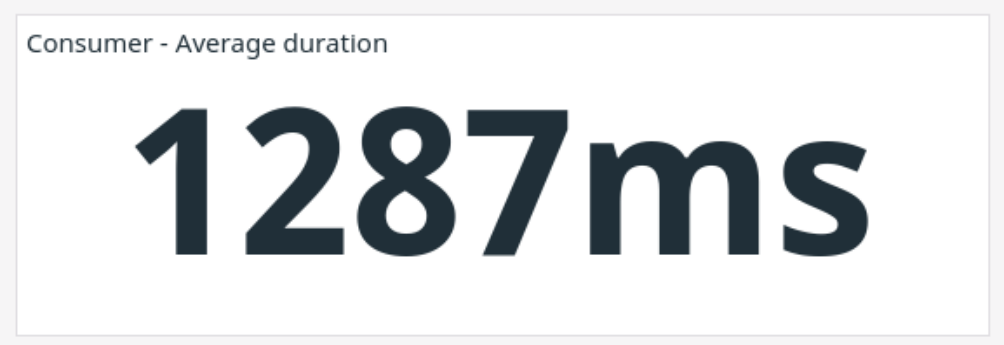
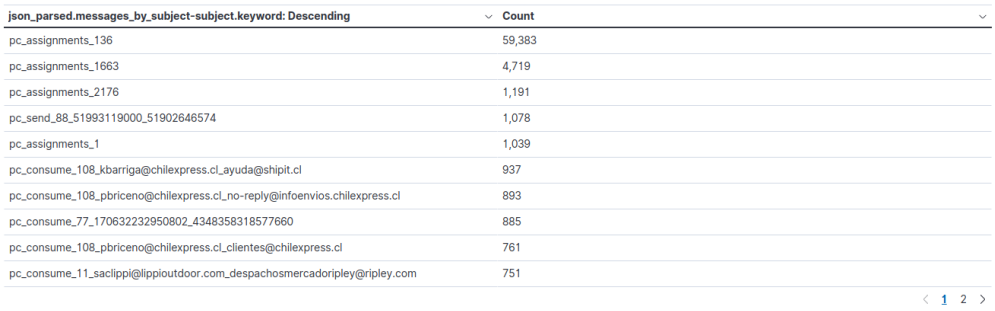


Figura 5.9: Foto con las métricas generadas en Datadog para el tiempo de ejecución de la lambda SQS Consumer posterior al despliegue de Chaski 2.0.



json_parsed.messages_by_subject-subject.keyword: Descending	Count
pc_assignments_136	59,383
pc_assignments_1663	4,719
pc_assignments_2176	1,191
pc_send_88_51993119000_51902646574	1,078
pc_assignments_1	1,039
pc_consume_108_kbarriga@chilexpress.cl_ayuda@shipit.cl	937
pc_consume_108_pbriceno@chilexpress.cl_no-reply@infoenvios.chilexpress.cl	893
pc_consume_77_170632232950802_4348358318577660	885
pc_consume_108_pbriceno@chilexpress.cl_clientes@chilexpress.cl	761
pc_consume_11_saclippi@lippioutdoor.com_despachosmercadoripley@ripley.com	751

1 2

Figura 5.10: Foto con las métricas generadas personalizadas para Chaski 2.0 para messages_by_subject posterior al despliegue de Chaski 2.0.

Top values subject	Average average request time (ms) ↓
pc_send_123_56971354232_56951974402	999.992
pc_consume_108_mortega@ext.chilexpress.cl_macarena.delrio@reversso.cl	999.988
pc_send_82_56957211492_56961315448	999.985
pc_consume_88_51993119000_51965118776	999.98
pc_consume_107_50671870541_50622074220	999.976
pc_send_82_56957211492_56992869922	999.974
pc_consume_221_56938629339_56949195078	999.972
pc_send_88_115004651862366_2665522256855879	999.969
pc_consume_1924_5214431378728_5217223032972	999.966
pc_consume_123_56971354232_56973049588	999.964
Other	392.643

Figura 5.11: Foto con las métricas generadas personalizadas para Chaski 2.0 para time_by_target posterior al despluegue de Chaski 2.0.

Las imágenes 5.7, 5.8, 5.9, 5.5 y 5.11 nos muestra el nivel de uso de Chaski 2.0 en relación a las invocaciones de lambdas, el comportamiento básico ya existente con Chaski 1.0 (uso de las colas) y con las nuevas funcionalidades, las métricas.

Las imágenes 5.5 y 5.11 muestran métricas de mensajes por subject y tiempo por subject que anteriormente no estaban disponibles. Esta situación da visibilidad de cuales son los clientes que usan más intensamente los recursos de Adereso Helpdesk, en estas imágenes el subject permite identificar a los clientes.

5.3.2. Manejo de errores

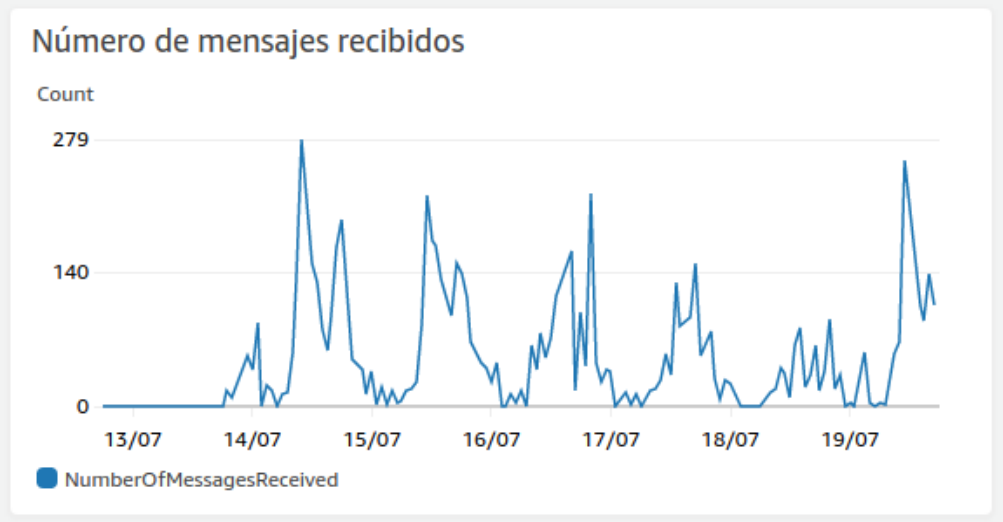


Figura 5.12: Foto con el gráfico de Amazon CloudWatch que muestra la cantidad de mensajes que ingresaron a la cola de errores.

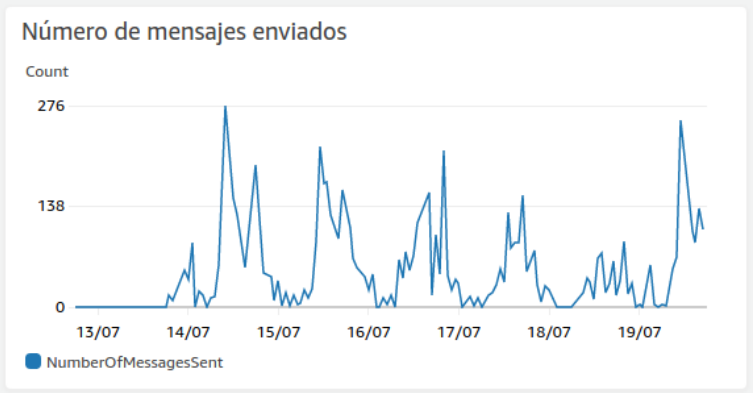


Figura 5.13: Foto con el gráfico de Amazon CloudWatch que muestra la cantidad de mensajes que salieron de la cola de errores.

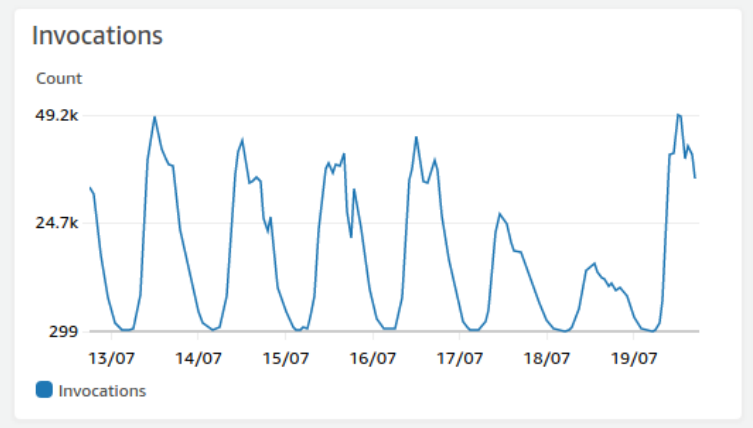


Figura 5.14: Foto con el gráfico de Amazon CloudWatch que muestra la cantidad de invocaciones de la lambda SQS Consumer.

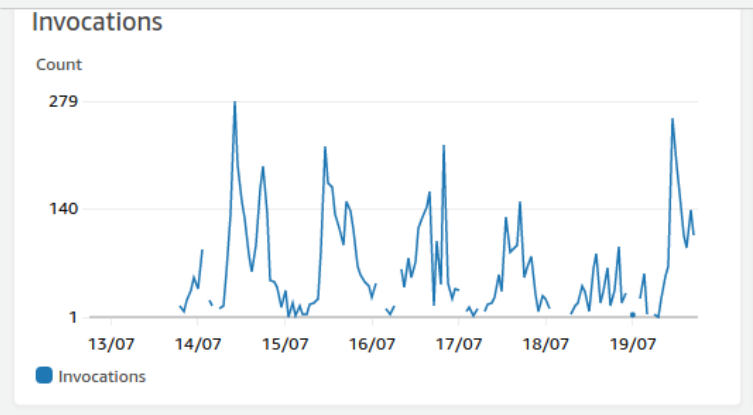


Figura 5.15: Foto con el gráfico de Amazon CloudWatch que muestra la cantidad de invocaciones de la lambda SQS Error Consumer.

Top values of json_parsed.error_by_subject-subject.keyword	Count of records
pc_consume_1050_191266410999385_4887890654570975	5
pc_consume_108_c8575810d016f5ecf9a5f3f2fe75bc13_18a1a915e16f39ab0491d16879d0c0a97f3ac2abe03aa551909e219c41d3579c	1
pc_consume_108_c8575810d016f5ecf9a5f3f2fe75bc13_f565e032a534792449ddada3644b02a60278c0da064e648623a0412ee0b30718	1
pc_consume_108_contactoemail@chilexpress.cl_luciaarriaza77@gmail.com	1
pc_consume_108_pbriceno@chilexpress.cl_angelica_nm2@hotmail.com	1
pc_consume_108_pbriceno@chilexpress.cl_matias.astorga.martinez@gmail.com	1
pc_consume_1204_5215534189427_5215522702604	5
pc_consume_123_56971354232_56930916229	1
pc_consume_123_56971354232_56983408719	6
pc_consume_123_56971354232_56989142230	5
Other	10,131

Figura 5.16: Foto con las métricas personalizadas generadas para Chaski 2.0 para error_by_subject posterior al despliegue de Chaski 2.0 para SQS Consumer y SQS Error Consumer.

date	timestamp	created_at	error	message
2021-07-17	2021-07-17 13:31:32	2021-07-17T13:35:38.167623	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_2059_1078
2021-07-17	2021-07-17 13:31:19	2021-07-17T13:35:25.234678	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_2059_1078
2021-07-17	2021-07-17 12:27:20	2021-07-17T12:31:27.872820	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_673_16394
2021-07-17	2021-07-17 12:26:39	2021-07-17T12:30:45.499274	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_673_16394
2021-07-17	2021-07-17 12:26:02	2021-07-17T12:30:10.621648	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_673_16394
2021-07-17	2021-07-17 12:25:51	2021-07-17T12:29:59.325666	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_673_16394
2021-07-17	2021-07-17 11:53:28	2021-07-17T11:57:34.859173	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_80_263370
2021-07-17	2021-07-17 11:52:47	2021-07-17T11:56:55.938982	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_18_439081
2021-07-17	2021-07-17 09:15:18	2021-07-17T09:19:22.494906	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_2059_1078
2021-07-17	2021-07-17 09:15:08	2021-07-17T09:19:15.652426	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_2059_1078
2021-07-17	2021-07-17 09:14:46	2021-07-17T09:18:53.490057	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_2059_1078
2021-07-17	2021-07-17 07:57:06	2021-07-17T08:01:12.035529	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_18_439081
2021-07-17	2021-07-17 07:49:41	2021-07-17T07:53:46.550452	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_2059_5461
2021-07-17	2021-07-17 07:48:58	2021-07-17T07:53:03.329212	ServerError(500, 'POST', 'https://api-cluster.postcenter.io/internal/v2/messa…	{"subject": "pc_consume_2059_5461

Figura 5.17: Foto con una muestra de los mensajes efectivamente siendo almacenados en DynamoDB.

Las imágenes 5.12, 5.13, 5.14, 5.15 y 5.16 nos muestran que el sistema de reintentos y de manejo de errores está funcionando como se espera. En particular la imagen 5.17 nos muestra que efectivamente hay una presencia de mensajes no recuperables. La mayor parte de estos errores corresponden a 2 clientes en mayor medida, el 2059 y el 80. Estas empresas corresponden a clientes que probaron Adereso Helpdesk, decidieron no usarlo pero no desconectan sus cuentas. Antes de la implementación en producción de Chaski 2.0 se sabía que existía una cantidad importante de errores por canales abandonados, pero gracias a Chaski 2.0 estos casos críticos fueron encontrados con mucha facilidad, cruzando datos generados por Chaski 2.0 con datos ya existentes en Adereso.

5.3.3. Seguimiento de mensajes

Para validar el seguimiento de mensajes se hizo una prueba única en producción. Se envió un mensaje por Messenger de Facebook a una cuenta conectada con Adereso Helpdesk. Se

comprobó que este mensaje efectivamente llegará al Helpdesk y finalmente se buscó en los registros de Chaski 2.0.

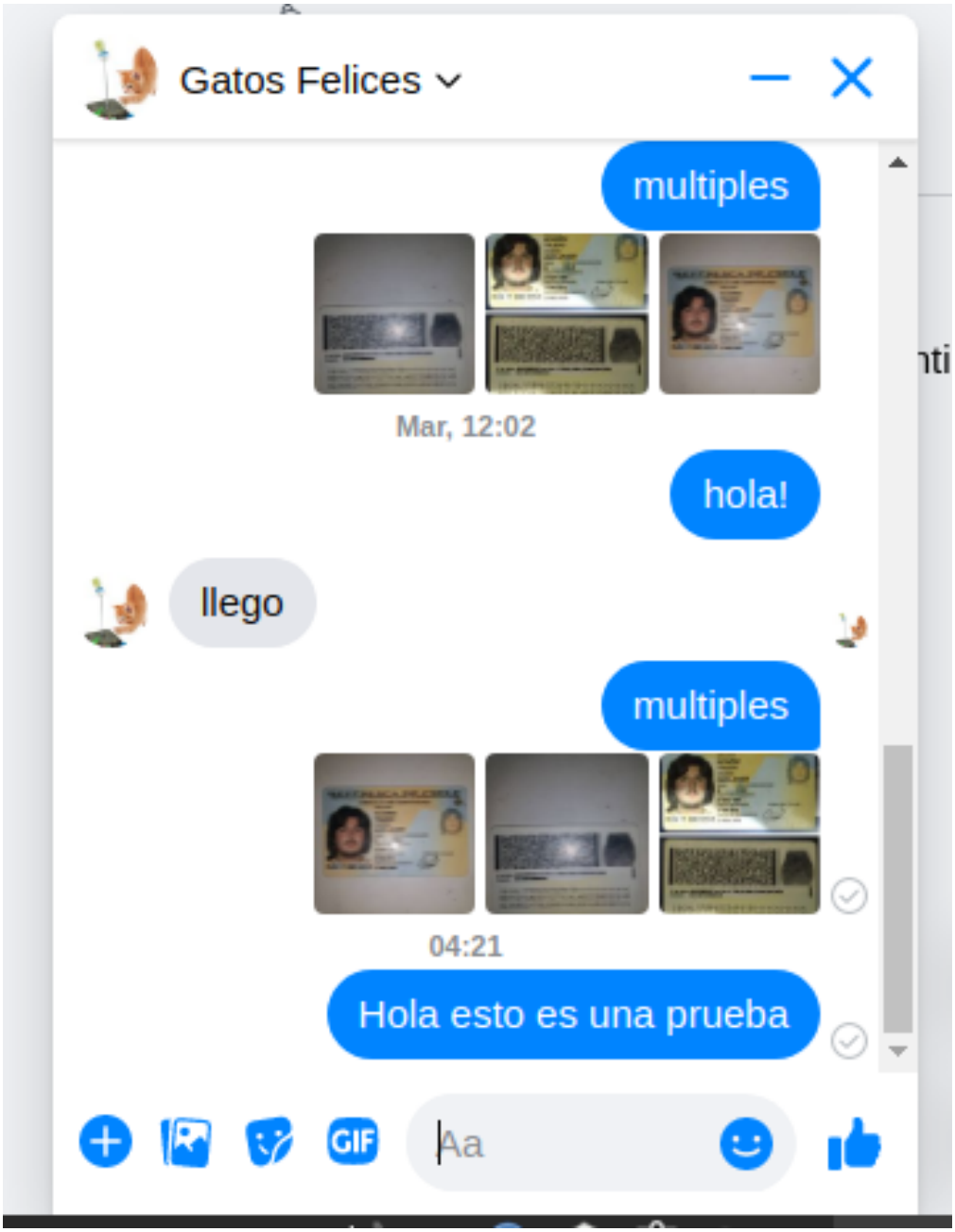


Figura 5.18: Foto de un mensaje siendo enviado a través de Facebook Messenger a un canal conectado a Adereso Helpdesk

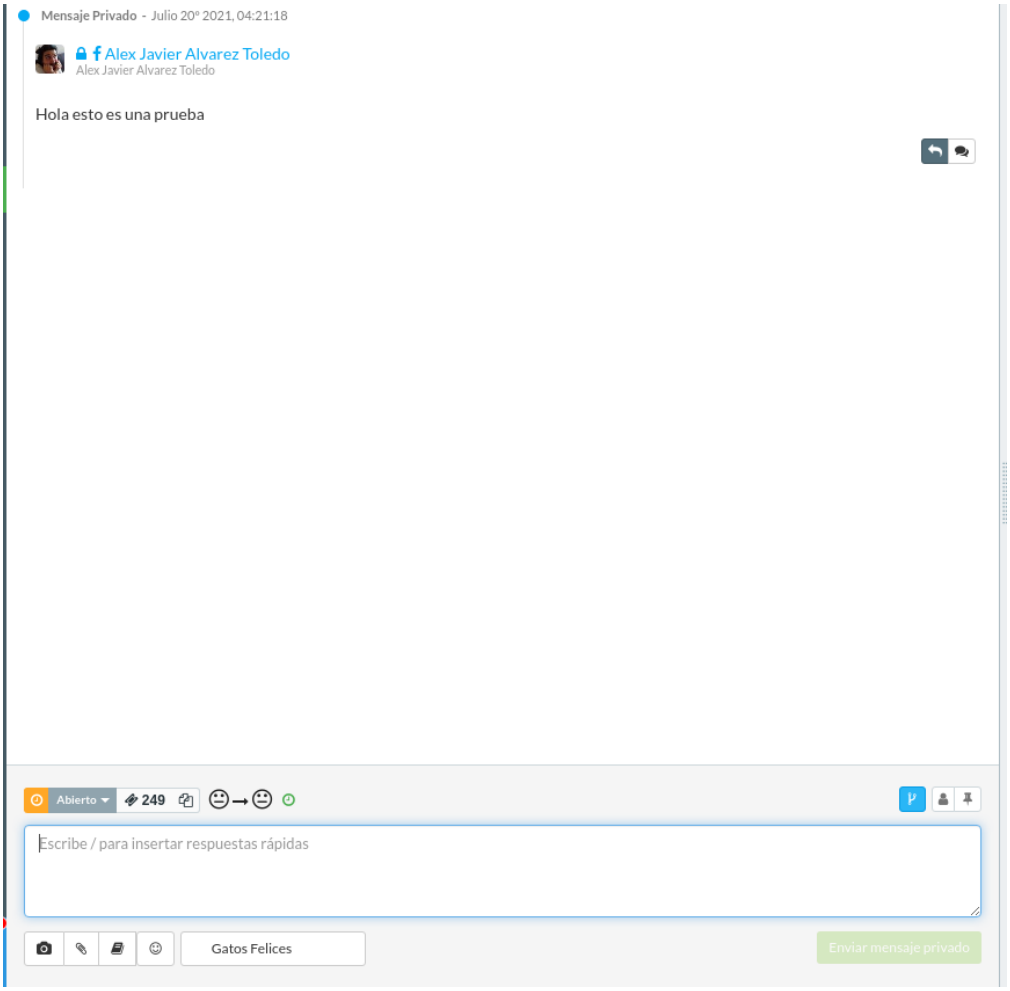


Figura 5.19: Foto del mensaje enviado previamente visible en el Adereso Helpdesk.

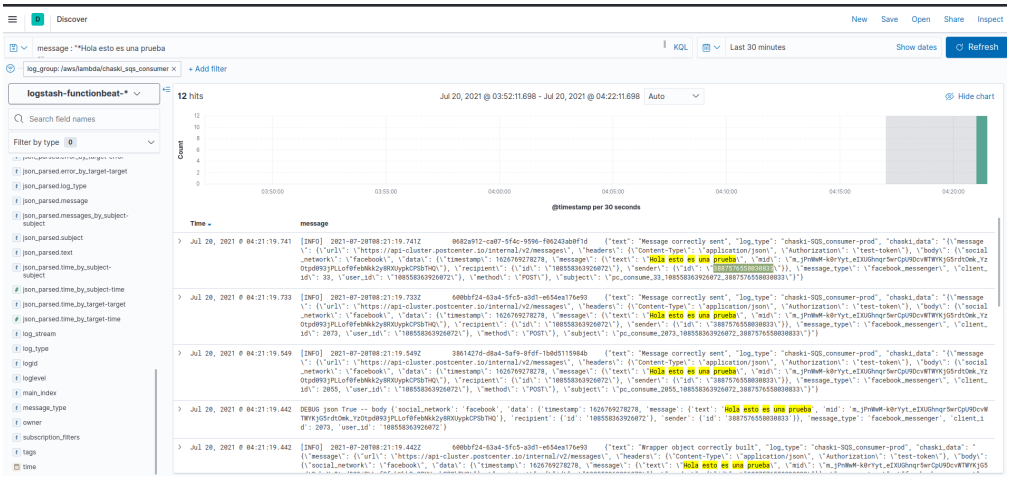


Figura 5.20: Foto con una muestra de cómo se ve el buscador de registros en Kibana.

La imagen 5.20 nos muestra de que manera, usando el contenido de un mensaje, se pueden buscar en Kibana los registros del procesamiento del mensaje en Chaski 2.0. Se comprueba así que el mensaje fue procesado en Chaski y que fue procesado con éxito. Esta herramienta permitirá a Adereso diagnosticar problemas previos o posteriores a Chaski con mayor facilidad, pudiendo determinar si el mensaje pasó por Chaski y si fue exitoso o no, además de establecer que error fue encontrado, de haberse producido uno. Por cómo se envían los mensajes al servicio ELK de Adereso desde Chaski 2.0 se puede hacer una búsqueda por varios parámetros, ejemplo, el identificar la cuenta en el proveedor. Para la prueba se usó el contenido del mensaje por comodidad.

Capítulo 6

Conclusiones

El éxito del proyecto Chaski 2.0 dependía no sólo de dar solución a los requisitos que fueron definidos, se debe también competir con la incertidumbre que genera en el equipo el querer reemplazar a servicios que llevan tiempo funcionando. Entendiendo que la necesidad de desarrollar un proyecto como Chaski 2.0 nace pues las características de los servicios utilizados no son suficientes para cumplir con la demanda del producto en que se usan, la confianza adquirida debido al tiempo que llevan funcionando no se puede ignorar. En la actualidad, debido al éxito de Chaski 2.0 la Adereso tiene el siguiente patrón de servicios. Chaski 2.0 es el principal servicio repartidor de mensajes de Adereso Helpdesk para canales de Facebook, Whatsapp, chat y email. Los otros canales aún no terminan su proceso de migración, pero eso escapa los alcances del proyecto. También, el servicio legacy continúa funcionando pero como un respaldo de Chaski 2.0, para dar continuidad ante potenciales fallas de Amazon y porque el canal de email puede recibir mensajes problemáticos para Chaski 2.0. Por limitaciones del tamaño de los datos que Chaski 2.0 puede recibir. Finalmente, el servicio Chaski 1.0 fue completamente reemplazado por Chaski 2.0, sin embargo parte importante de la arquitectura de Chaski 2.0 se extiende del proyecto original. Pero como producto solo está funcionando la nueva versión.

6.1. Resumen del trabajo realizado

El trabajo realizado se puede resumir en tres necesidades por satisfacer: la de un servicio robusto, un servicio monitoreable y la persistencia de los mensajes. En lo que a robustez respecta se desarrollaron librerías que reducen duplicación de código, se escribieron pruebas unitarias que toman en cuenta la duplicación que no se pudo evitar y que dan cuenta del correcto comportamiento esperado, y se desarrolló un sistema de reintentos para asegurar que se realicen todos los esfuerzos para que un mensaje llegue a su destino.

Con respecto al monitoreo se suplementó el servicio Datadog de métricas y dashboards con el servicio ELK de Adereso, agregando nuevas métricas personalizadas, nuevos dashboards y un sistema de exploración de los registros de la aplicación.

Finalmente, para asegurar la persistencia de los datos se desarrolló el sistema de reintentos y una base de datos para guardar mensajes que no pudieron ser entregados exitosamente. Los datos de aquellos mensajes correctamente entregados se consideran ya persistentes, pero los que fallaron en su procesamiento deben guardarse para asegurar que ningún dato se pierda.

6.2. Grado de cumplimiento de los objetivos

Con respecto al primer objetivo especifico: Asegurar el seguimiento de al menos el 99 % de los mensajes enviados por Adereso Helpdesk al servicio Chaski. En la sección de validación se mostró un experimento realizado en el ambiente de QA que buscó demostrar que el funcionamiento de Chaski 2.0 es el esperado. Uno de los puntos importantes de este experimento es evidenciar que se genera la cantidad esperada de mensajes y métricas. Este experimento se consideró exitoso, todos los mensajes esperados fueron recibidos de manera correcta y las métricas esperadas fueron generadas con un 8 % (48 mensajes extra de 600 esperados) de falla, dentro de los parámetros esperados. También en la sección de validación se mostró un ejemplo de cómo a partir del contenido de un mensaje se pudieron encontrar los registros de su paso por las lambdas de Chaski 2.0, y se ve como llega exitosamente a Adereso Helpdesk. Es importante mencionar que el error en las métricas de la lambda SQS Error Consumer se consideró una falla secundaria para la empresa. Además de la cantidad de errores registrado estuvo dentro de parámetros tolerables, el objetivo principal es que los mensajes lleguen a su destino o se almacenen. Sin embargo, se debe trabajar en encontrar la causa de esta discrepancia.

Con respecto al segundo objetivo especifico: Asegurar que en caso de fallas que produzcan pérdidas de mensajes se mantenga registro de al menos el 90 % de los datos. En la sección de validación, en el experimento realizado en el ambiente de QA también se buscaba evaluar el almacenamiento de los mensajes. Se esperaba que de los doscientos mensajes, 150 (los que se enviaron a endpoints que generaban errores) fueran almacenados. Esto se cumplió en un 100 %. También se mostró como en el ambiente de producción se están guardando los errores exitosamente, sin embargo, como el ambiente de producción no es controlado, no se puede dar certeza del porcentaje de efectividad. Se necesitan desarrollar mejores sistemas de monitoreo en los otros sistemas de Adereso Helpdesk para poder generar esta información.

En lo que a los tres últimos objetivos específicos respecta: desarrollar sistema de reintentos de envío de mensajes; mejorar el sistema de monitoreo, considerando métricas de origen y destino; mejorar el sistema de logs para facilitar la detección de problemas. Se mostraron en la sección de validación las métricas propuestas. El hecho de que la lambda de SQS Error Consumer se esté ejecutando, pero en menor medida que la lambda SQS Consumer, nos permite concluir que hay mensajes que generan errores y se están procesando. Finalmente también se mostró que la base de datos de errores se está poblando con mensajes.

Los objetivos secundarios no se lograron abordar. Pero considero importante mencionar que efectivamente la lambda SQS Consumer está funcionando con menores tiempos de ejecución, lo que se puede observar al comparar las fotos 3.6 y 5.9. Sin embargo, este incremento en eficiencia no tiene que ver con el desarrollo de Chaski 2.0. Efectivamente se hicieron pequeñas optimizaciones, pero el factor limitante es la consulta HTTP. La mejora en el rendimiento

de la lambda observado en SQS Consumer es producto de una mejora en el rendimiento en general de la plataforma de Adereso Helpdesk.

6.3. Análisis de los resultados y el impacto del proyecto

El proyecto Chaski 2.0 ya fue puesto en producción en la empresa Adereso. En lo que respecta al problema que se busca resolver, repartir mensajes desde productores a consumidores, Chaski 2.0 cumple con su requisito funcional completamente.

Chaski 2.0 generó valor para la empresa en el día que fue puesta en producción, ya se conocía la presencia de errores producto de configuraciones en los canales conectados, pero no se había podido identificar fácilmente los canales o clientes que generaban estos errores. Gracias a las herramientas de monitoreo entregadas en el proyecto, se detectó una alta presencia de errores en la entrega de mensajes, estos estaban concentrados en unos pocos "subjects" y direcciones objetivo. Con esta información se pudieron identificar los canales problemáticos y solucionar la situación.

También, en la medida que más componentes de Adereso Helpdesk se transformen en microservicios mayor importancia cobrará el repartidor de mensajes, tener un sistema con alta observabilidad y confianza es fundamental en el plan de cambio arquitectónico de la empresa. Es importante notar que Chaski no es el único canal de comunicación interno de la aplicación, pero gracias al trabajo desarrollado no sería el eslabón débil de la cadena productiva.

6.4. Aprendizajes del proceso de desarrollo

Un pequeño detalle ignorado en la especificación de requerimientos y el diseño de la solución, como el no considerar la retrocompatibilidad de la solución con los servicios que ya están siendo usados en producción, llevó a una incompatibilidad en el despliegue del proyecto que causó un atraso significativo de la entrega.

Una conclusión, y aprendizaje de esta experiencia, es que todo diseño de una solución debería partir definiendo cómo se hará el despliegue de los servicios y cómo va a afectar el resto del proceso productivo que ya está funcionando, de existir. Es muy sencillo asumir que solo el diseño funcional es lo importante y que mientras se respeten las API todo va a funcionar, pero pueden haber condiciones de carrera, o la api puede no aguantar la nueva carga, o, como en el caso de Chaski, la solución no es retrocompatible y demanda un downtime del servicio si se quiere implementar. No importa lo bien que el software resuelva el problema si las dificultades de implementación hacen que no se pueda usar.

6.4.1. Planificación ideal en retrospectiva

Siguiendo la planificación del proyecto como un solo gran entregable que se desarrolló en etapas, que se planificó para desarrollar el trabajo en forma divida pero que no contemplaba entregas parciales, se logró alcanzar los objetivos propuesto para el proyecto Chaski 2.0. Sin embargo, esta forma de trabajo no fue la ideal, la principal dificultad derivada del patrón de entrega usado fue que el despliegue fue mas complejo de lo necesario.

En retrospectiva, de haberse realizado entregas continuas de los avances parciales del proyecto, el problema en el despliegue del mismo habría sido menor y más fácil de corregir. Debido a que el punto de conflicto habría ocurrido sobre un código ya entregado, el error habría sido fácil de identificar y el alcance de las modificaciones habrían sido limitados. Es importante mencionar que la restricción de entregar el proyecto como un gran paso de despliegue fue completamente innecesaria.

La planificación ideal que se debió haber seguido con el fin de permitir generar entregas continuas es:

1. Librería chaski requests
2. Librería dynamo handler
3. Chaski logger
4. Message wrapper
5. Métricas

La librería Chaski Requests es la primera pues se puede implementar sin alterar en absoluto el comportamiento de Chaski, es un wrapper que permite una definición interna de cómo se manejan las respuestas de los request http hechos en el código.

Se sigue con dynamo handler pues es una librería que se inserta en respuesta a las definiciones de errores nuevas definidas en el paso anterior. En este punto Chaski cumple con el requerimiento de persistencia de la información, y estaríamos cumpliendo con uno de los requerimientos importantes del proyecto.

Se sigue con chaski logger por simplicidad, ya entregado el proyecto sabemos que la mejor solución para Adereso era usar un dump de los logs desde Cloud Watch a Logstash, y el servicio ELK de adereso se desarrolló en la empresa en paralelo a las mejoras de Chaski, pero el rediseño de los mensajes de logs podría haberse hecho en cualquier punto. En este momento mejoramos la mantenibilidad del proyecto, otro requisito importante.

En este punto ya se habían entregado la mitad de los requerimientos de la empresa potencialmente en el servicio de producción. Y se comenzará a desarrollar el procesamiento paralelo de mensajes con errores y el protocolo de reintentos. Este es el desarrollo más complejo y el que produjo los problemas de retrocompatibilidad.

Se debió haber desarrollado el servicio de métricas personalizado al final, cumpliendo así con el último requerimiento de la empresa. El servicio de métricas es una extensión de la

librería Chaski Logger por lo que se debe desarrollar después de dicha librería, pero la razón de dejarlo para el final es que una vez terminado todo el código se pudo determinar con claridad donde encontramos los puntos en los cuales es importante generar métricas, antes de tener el proyecto completo habían ideas, pero no certezas.

6.5. Beneficios de las arquitecturas basadas en Cloud Computing

En la literatura podemos leer sobre los beneficios y desventajas de utilizar una arquitectura basada en servicios serverless. Pero en la práctica la decisión de seguir o no cierto paradigma no se basa en todas sus cualidades, algunas características toman precedencia y se vuelven indispensables. En particular en este caso se identificaron dos beneficios irrenunciables de usar arquitectura serverless en el proyecto Chaski, y que configuran una buena razón para optar por este tipo de productos en futuros proyectos: el bajo costo de inversión para levantar un prototipo y la claridad al momento de calcular los costos de levantamiento y mantención de un nuevo servicio.

6.5.1. Bajo costo del prototipado

Una de las características más importantes que se requerian del proyecto Chaski era que el servicio fuese escalable. Usar servicios de procesamiento en la nube simplificó este requisito, gracias a esta característica clásica de los servicios de Cloud Computing se pudo diseñar un proyecto que solo se centra en dar solución al problema de la manera más simple posible, con la confianza de que soluciones ineficientes incurren en mayores costos de manera lineal con respecto a la cantidad de memoria, el tiempo de ejecución y la cantidad de instancias de ejecución. Y por lo tanto una segunda etapa de desarrollo se puede centar en mejorar el rendimiento en cada uno de estos puntos.

Todas estas características permiten ahorrar al momento de desarrollar prototipos pues: no se debe incurrir en el costo de nuevo hardware, el trabajo desarrollado genera un producto utilizable en producción (posiblemente uno ineficiente, pero usable), el problema de disponibilidad se encuentra resuelto por el proveedor de servicios cloud, por otro lado, descartar el proyecto implica solo dejar de consumir los servicios utilizados. Lo más relevante de todo esto es que todos estos beneficios no conllevan a un mayor tiempo de desarrollo por parte del equipo.

Entender las particularidades de desarrollar un buen servicio en la nube lleva tiempo y tiene un costo, pero no es requisito para poder desarrollar un proyecto y montarlo, es deseable pero, la solución puede ser ineficiente y aún así funcionar. Más aún que el ahorro en hardware, la principal reducción de costos de un prototipo montado en la nube se da en la explotación del acceso y alta disponibilidad de los recursos que debe asegurar el proveedor de servicios cloud para asegurar la disponibilidad, escalabilidad y acceso del producto que se está desarrollando en la plataforma cloud utilizada.

6.5.2. La clara definición de costos

Al momento de decidir si invertir en un proyecto es de suma importancia poder determinar cuánto se debe gastar para ponerlo en marcha y para mantenerlo en servicio. En el caso de usar servidores propios, o incluso utilizar un patrón BAAS, el cálculo de estos costos se vuelven complejos, ya que determinar que máquina conviene usar, para que carga, cuanto cuesta comprarla, cuánto podemos obtener al liquidar el hardware, todas estas son preguntas que se deben responder para definir un correcto perfil de costos del proyecto.

Al utilizar un paradigma de FAAS al diseñar el proyecto Chaski se pudo determinar clara y fácilmente cuánto se debía invertir inicialmente por el proyecto y cuánto se gastará por día para mantenerlo funcionando. Es de suma importancia recalcar que el beneficio no es en términos de menores costos, es en acceso a información clara sobre los gastos.

6.6. Ideas para el futuro de Chaski

Existen dos principales ideas sobre cómo seguir trabajando en mejorar Chaski en el futuro. La primera tiene que ver con seguir mejorando el rendimiento actual del servicio. La segunda está relacionada con extender el uso de la plataforma.

6.6.1. Mejorando el servicio

El principal desarrollo pendiente en el proyecto fue aumentar la eficiencia del servicio Chaski. El trabajo se centró en la efectividad de la solución, asumiendo que las optimizaciones son un tema una vez que la solución cumple con lo deseado. En la actualidad se entiende que la operación más cara e ineficiente es la consulta HTTP final del procesamiento, la solución propuesta debe centrarse en encontrar algo que hacer con los recursos de Chaski mientras se espera por la respuesta del servidor remoto.

El segundo desarrollo respecto al uso actual de Chaski es implementar nuevas métricas. Los resultados obtenidos con las métricas ya implementadas fueron destacables, pero lo primero que ocurrió luego de observarlas fue que se pensaron en nuevas métricas aún más útiles para describir el funcionamiento de la plataforma. En particular se van a implementar métricas en función del identificador de cliente en lugar de subject para mensajes, errores y tiempos.

6.6.2. Nuevos usos de Chaski

Chaski es un repartidor de mensajes que se pensó para solucionar el problema de comunicar los servicios generadores de mensajes con los servidores de Adereso Helpdesk. La solución está limitada por las características necesarias para satisfacer esta necesidad, pero se podría modificar parcialmente para dar solución a otros problemas de conectividad en la

empresa. En este sentido el primer desarrollo importante que permitirá probar Chaski en otros contextos es el de automatizar el despliegue de los servicios que se usan, para esto se está pensando usar Serverless Framework, una herramienta que permite organizar y ejecutar el despliegue de aplicaciones serverless que usan múltiples servicios.

Bibliografía

[1] Adereso. Adereso como funciona [en línea], disponible en `https://www.adere.so/como-funciona` [visitado: 16-07-2021].

[2] Adereso. Adereso landing page [en línea], disponible en `https://www.adere.so/` [visitado: 16-07-2021].

[3] Omar Al-Debagy and Peter Martinek. A comparative review of microservices and monolithic architectures. *CoRR*, abs/1905.07997, 2019.

[4] Amazon. Amazon api gateway [en línea], disponible en `https://aws.amazon.com/es/api-gateway/` [visitado: 16-07-2021].

[5] Amazon. Amazon cloudwatch [en línea], disponible en `https://aws.amazon.com/es/cloudwatch/` [visitado: 16-07-2021].

[6] Amazon. Amazon dynamodb [en línea], disponible en `https://aws.amazon.com/es/dynamodb/` [visitado: 16-07-2021].

[7] Amazon. Amazon simple queue service [en línea], disponible en `https://aws.amazon.com/es/sqs/` [visitado: 16-07-2021].

[8] Amazon. Aws lambda [en línea], disponible en `https://aws.amazon.com/es/lambda/` [visitado: 16-07-2021].

[9] Amazon. Aws well-architected. frameworkconcepts. canary deployment [en línea], disponible en `https://wa.aws.amazon.com/wellarchitected/2020-07-02T19-33-23/wat.concept.canary-deployment.en.html` [visitado: 16-07-2021].

[10] Amazon. Características de amazon api gateway [en línea], disponible en `https://aws.amazon.com/es/api-gateway/features/` [visitado: 16-07-2021].

[11] Amazon. Características de amazon dynamodb [en línea], disponible en `https://aws.amazon.com/es/dynamodb/features/` [visitado: 16-07-2021].

[12] Amazon. Características de amazon lambda [en línea], disponible en `https://aws.amazon.com/es/lambda/features/` [visitado: 16-07-2021].

[13] Amazon. Características de amazon sqs [en línea], disponible en `https://aws.amazon.com/es/sqs/features/` [visitado: 16-07-2021].

[14] Amazon. Core components of amazon dynamodb [en línea], disponible en https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html [visitado: 16-07-2021].

[15] Amazon. Netflix en aws [en línea], disponible en https://aws.amazon.com/es/solutions/case-studies/netflix/ [visitado: 16-07-2021].

[16] Amazon. Precios de amazon api gateway [en línea], disponible en https://aws.amazon.com/es/api-gateway/pricing/ [visitado: 16-07-2021].

[17] Amazon. Precios de amazon dynamodb [en línea], disponible en https://aws.amazon.com/es/api-gateway/pricing/ [visitado: 16-07-2021].

[18] Amazon. Precios de aws lambda [en línea], disponible en https://aws.amazon.com/es/lambda/pricing/ [visitado: 16-07-2021].

[19] Apache. Apache kafka landing page [en línea], disponible en https://kafka.apache.org/ [visitado: 16-07-2021].

[20] Apache. Kafka protocol guide [en línea], disponible en https://kafka.apache.org/protocol [visitado: 16-07-2021].

[21] Vasanth Asokan, Ludovic Galibert, and Sangeeta Narayanan. Developer experience lessons operating a serverless-like platform at netflix [en línea], disponible en https://netflixtechblog.com/developer-experience-lessons-operating-a-serverless-like-platform-at-netflix-a8bb [visitado: 16-07-2021].

[22] Atlassian. Software testing for continuous delivery. what is software testing? [en línea], disponible en https://www.atlassian.com/continuous-delivery/software-testing [visitado: 16-07-2021].

[23] Atlassian. What is continuous integration? [en línea], disponible en https://www.atlassian.com/continuous-delivery/continuous-integration [visitado: 16-07-2021].

[24] Datadog. Datadog caracteristicas del producto [en línea], disponible en https://www.datadoghq.com/product/ [visitado: 16-07-2021].

[25] Datadog. Datadog landing page [en línea], disponible en https://www.datadoghq.com/ [visitado: 16-07-2021].

[26] Biblioteca del Congreso Nacional de Chile. Ley fácil, guía legal sobre: Estatuto de las pymes [en línea], disponible en https://www.bcn.cl/leyfacil/recurso/estatuto-de-las-pymes [visitado: 16-07-2021].

[27] Celery development team. Introduction to celery [en línea], disponible en https://docs.celeryproject.org/en/stable/getting-started/introduction.html [visitado: 16-07-2021].

[28] IBM Cloud Education. Microservices [en línea], disponible en https://www.ibm.com/cloud/learn/microservices [visitado: 16-07-2021].

[29] Elastic. Logstash landing page [en línea], disponible en https://www.elastic.co/es/logstash/ [visitado: 16-07-2021].

[30] Elastic. ¿qué es el elk stack? [en línea], disponible en https://www.elastic.co/es/what-is/elk-stack [visitado: 16-07-2021].

[31] Elastic. ¿qué es kibana? [en línea], disponible en https://www.elastic.co/es/what-is/kibana [visitado: 16-07-2021].

[32] Ludovic Galibert, Vasanth Asokan, and Sangeeta Narayanan. Developer experience lessons operating a serverless-like platform at netflix — part ii [en línea], disponible en https://netflixtechblog.com/developer-experience-lessons-operating-a-serverless-like-platform-at-netflix-part [visitado: 16-07-2021].

[33] Gartner. Gartner glossary, scalability [en línea], disponible en https://www.gartner.com/en/information-technology/glossary/scalability [visitado: 16-07-2021].

[34] GitHub. Githubdocs about pull requests [en línea], disponible en https://docs.github.com/en/github/collaborating-with-pull-requests/proposing-changes-to-your-work-with-pull-requests/about-pull-requests [visitado: 16-07-2021].

[35] Google. Cloud functions precios [en línea], disponible en https://cloud.google.com/functions/pricing [visitado: 16-07-2021].

[36] Google. Cloud run [en línea], disponible en https://cloud.google.com/run [visitado: 16-07-2021].

[37] Google. Cloud run precios [en línea], disponible en https://cloud.google.com/run/pricing [visitado: 16-07-2021].

[38] Google. Cloud sql [en línea], disponible en https://cloud.google.com/sql [visitado: 16-07-2021].

[39] Google. Descripción general de cloud functions [en línea], disponible en https://cloud.google.com/functions/docs/concepts/overview [visitado: 16-07-2021].

[40] Google. Documentación de cloud functions [en línea], disponible en https://cloud.google.com/functions/docs [visitado: 16-07-2021].

[41] Google. Precios de cloud sql [en línea], disponible en https://cloud.google.com/sql/pricing [visitado: 16-07-2021].

[42] IBM. Blog. serverless compiting [en línea], disponible en https://www.ibm.com/cloud/learn/serverless [visitado: 16-07-2021].

[43] IBM. Blog. software development [en línea], disponible en https://www.ibm.com/topics/software-development [visitado: 16-07-2021].

[44] Peter Mell and Timothy Grance. The nist definition of cloud computing. *NIST Special Publication 800145*, 2021.

[45] Microsoft. Microsoft docs, microservices architecture style [en línea], disponible en https://docs.microsoft.com/en-us/azure/architecture/guide/architecture-styles/microservices [visitado: 16-07-2021].

[46] Microsoft. Microsoft docs, unit test basics [en línea], disponible en https://docs.microsoft.com/en-us/visualstudio/test/unit-test-basics?view=vs-2019 [visitado: 16-07-2021].

[47] Gabriel Moskovicz. Elasticsearch: Primeros pasos [en línea], disponible en https://www.elastic.co/es/webinars/getting-started-elasticsearch [visitado: 16-07-2021].

[48] MQTT. Mqtt: The standard for iot messaging [en línea], disponible en https://mqtt.org/ [visitado: 16-07-2021].

[49] Neha Narkhede, Gwen Shapira, and Todd Palino. *Kafka The Definitive Guide*. O'Reilly, 2017.

[50] Netflix. What is netflix? [en línea], disponible en https://help.netflix.com/en/node/412 [visitado: 16-07-2021].

[51] Oasis. Iso and iec approve oasis amqp advanced message queuing protocol [en línea], disponible en https://www.oasis-open.org/news/pr/iso-and-iec-approve-oasis-amqp-advanced-message-queuing-protocol/ [visitado: 16-07-2021].

[52] Oracle. 10 high availability concepts and best practices [en línea], disponible en https://docs.oracle.com/cd/A91202_01/901_doc/rac.901/a89867/pshavdtl.htm [visitado: 16-07-2021].

[53] Oracle. Deploying weblogic platform applications. deployment checklists [en línea], disponible en https://docs.oracle.com/cd/E13196_01/platform/docs81/deploy/checklist.html [visitado: 16-07-2021].

[54] Oracle. The producer/consumer problem [en línea], disponible en https://docs.oracle.com/cd/E19455-01/806-5257/sync-31/index.html [visitado: 16-07-2021].

[55] STEN PITTET. Atlassian ci/cd, an introduction to code coverage [en línea], disponible en https://www.atlassian.com/continuous-delivery/software-testing/code-coverage [visitado: 16-07-2021].

[56] RabbitMQ. Rabbitmq configuration overview [en línea], disponible en https://www.rabbitmq.com/configure.html [visitado: 16-07-2021].

[57] RabbitMQ. Rabbitmq landing page. rabbitmq is the most widely deployed open source message broker [en línea], disponible en https://www.rabbitmq.com/ [visitado: 16-07-2021].

[58] Vikram Sreekanti, Chenggang Wu, Xiayue Charles Lin, Johann Schleier-Smith, Joseph Gonzalez, Joseph M. Hellerstein, and Alexey Tumanov. Cloudburst: Stateful functions-as-a-service. *Proc. VLDB Endow.*, 13(11):2438–2452, 2020.

[59] Davide Taibi, Josef Spillner, and Konrad Wawruch. Serverless computing-where are we now, and where are we heading? *IEEE Softw.*, 38(1):25–31, 2021.

[60] Stomp Development Team. Stomp the simple text oriented messaging protocol [en línea], disponible en `http://stomp.github.io/` [visitado: 16-07-2021].