



Instituto Tecnológico de Buenos Aires

72.40 - Ingeniería del Software II

Final – Analítico Web

Profesores:

Juan Martín Sotuyo, Guido Matías Mogni

Alumnos:

Federico Shih 62293,

Franco Rupnik 62032,

Matías Manzur 62498,

Santino Ranucci 62092,

Mauro Báez 61747

Fecha de Entrega:

Miércoles 13 de Diciembre 2023

1. Índice

1. Índice	2
2. Introducción y consigna	4
2.1 Consigna: Sistema 4 - Analíticos Web	4
2.2 Introducción	4
3. Requerimientos	5
3.1. Requerimientos funcionales	5
3.2. Requerimientos no funcionales	5
3.3. Usuarios	6
3.4. Casos de uso	6
4. Atributos de calidad	8
4.1. Scalability	8
4.2. Availability	8
4.3. Customizability	8
4.4. Performance	8
4.5. Security	9
4.6. Usability	9
5. Vista Física	10
6. Arquitectura Propuesta	11
6.1. Arquitectura de microservicios	12
6.1.1. Load Balancing y autoscaling	12
6.1.2. Microservicio de usuarios, organizaciones y roles	13
6.1.3. Microservicio de configuración de proyectos	13
6.1.4. Microservicio de templates	14
6.1.5. Pulsar Worker Queue Cluster	14
6.1.6. Data cleaning y Data enrichment pipeline.	15
6.1.7. Microservicio de reporting y analytics.	15
6.2. Load balancers externos	15
6.3. Cassandra con Apache Spark	16
6.4. Estructura en Cassandra	17
6.5. Spark Jobs	18
6.6. Apache Pulsar	18
6.7. Events API	18
6.8. SDKs	20
6.8.1. Kotlin SDK, Java SDK y Swift SDK	20
6.8.2 Javascript SDK	21
6.9. Templates.	21
6.10. API Gateway	22
6.11. SPA	22

6.12. PostgreSQL	23
7. Puntos críticos del sistema	24
7.1. Recolección de Datos	24
7.2. Procesado de Datos para Reportes	24
7.3. Generación y Visualización de Reportes	24
7.4. Acceso a los microservicios	24
8. Escenarios	26
8.1. Scalability	26
8.2. Availability	26
8.3. Customizability	27
8.4. Performance	27
8.5. Security	28
8.6. Usability	29
9. Supuestos, riesgos, no riesgos y trade-offs	30
10. Bibliografía	32

2. Introducción y consigna

2.1 Consigna: Sistema 4 – Analíticos Web

Se pide una plataforma de web analytics, similar a “Google Analytics”. El sistema debe permitir levantar eventos de páginas webs (clicks, búsquedas, etc) así como otros datos de los navegadores para tomar estadísticas (locale, versión, etc). No existe límite para estos datos, ya que los navegadores proporcionan cada vez más información.

A partir de los datos obtenidos, el sistema debe permitir la realización de reportes de variada complejidad, con cantidades crecientes de información. Estos reportes serán realizados por el usuario final que no es técnico sino especialista en marketing, y dependerá de los eventos de cada página, con lo que no pueden ser predefinidos.

La plataforma debe proveer una API que permita a otros sistemas integrarse para enviar eventos, así como un SDK para móviles. De más está decir que la escalabilidad es primordial.

2.2 Introducción

El informe busca desarrollar una arquitectura que permite levantar un servicio de analítica, minimizando riesgos, detallando compromisos y maximizando beneficios y durabilidad.

Se analizaron features actuales de Google Analytics y cualidades notadas por la consigna para obtener el listado de funcionalidades y atributos de calidad que más cumplían con el proyecto. Se definen los usuarios que interactúan con el sistema y sus escenarios de uso. Después se crea un diagrama de la arquitectura, con detalles como deployment, lenguajes de programación y frameworks usados.

Finalmente se analizan los distintos escenarios y cómo son resueltos con la arquitectura propuesta, además de supuestos tomados, riesgos, no riesgos y la bibliografía que se utiliza para justificar nuestras decisiones.

3. Requerimientos

3.1. Requerimientos funcionales

- 1) Recolectar eventos de clicks y carga con datos pertinentes del usuario desde una página web o aplicación móvil.
- 2) Permitir la creación de reportes analíticos de variada complejidad sobre los datos recolectados, permitiendo agregar templates de gráficos acorde a sus preferencias.
- 3) Habilitar la visualización de los reportes creados.
- 4) Facilitar la segmentación y filtrado de datos recolectados por tags predefinidos y personalizados (franja horaria, mercado, país, demografía).
- 5) Permitir la creación de un proyecto (analítica de una página web), así como el registro e inicio de sesión de usuarios para acceder a los reportes del mismo.
- 6) Permitir la configuración de permisos y restricciones sobre proyectos de una organización.
- 7) Permitir que el usuario pueda configurar lógica de filtrado (drop de eventos) simple en base a propiedades de eventos (hostname, ruta, propiedades personalizadas).

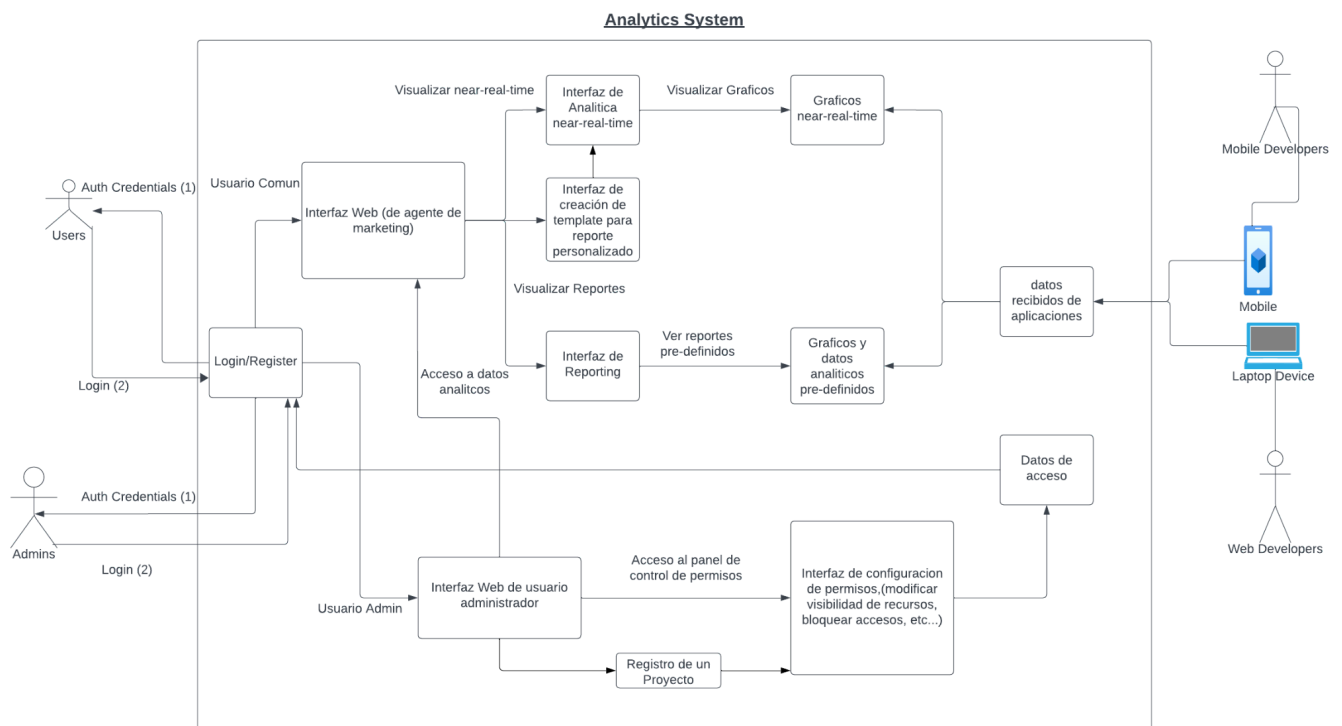
3.2. Requerimientos no funcionales

- El sistema necesita estar disponible 24/7 para capturar la mayor cantidad de eventos y visualizar reportes en todo momento.
- El sistema debe tolerar picos de demanda de consumición de datos sin degradar la performance del mismo.
- El sistema debe soportar una cantidad ilimitada de datos recibidos de los navegadores.
- El sistema debe permitir control de acceso de usuarios, autenticación y manejo de permisos de lectura, creación de reportes y configuración por cada campaña.
- El sistema debe responder a pedidos de visualización de reportes analíticos en un máximo de 5 segundos.
- El sistema debe permitir realizar tareas complejas de analítica (operaciones group by, proyecciones, uniones entre orígenes de datos).
- El sistema debe permitir de manera sencilla la elección de qué datos son pertinentes para los reportes personalizados.
- El sistema debe permitir visualizar datos a tiempo cercano al real.

3.3. Usuarios

- El desarrollador de la aplicación web y/o móvil que configura la recolección de eventos generados por los usuarios en la aplicación.
- El especialista en marketing que consulta los reportes.
- Un administrador de la empresa que maneja los permisos de usuario de los especialistas de marketing.

3.4. Casos de uso



Uso de agentes de marketing:

Los agentes de marketing, al iniciar sesión o registrarse en la plataforma, obtienen acceso a una interfaz web especializada para potenciar sus capacidades y optimizar sus estrategias comerciales. Mediante esta interfaz, pueden aprovechar una variedad de distintas funcionalidades estratégicas.

Pueden visualizar e interpretar gráficos analíticos que muestran la actividad de los clientes en las aplicaciones web y móviles. Así pudiendo identificar tendencias en actividad, sectores menos visitados de la aplicación o secciones con mayor interacción. Estos gráficos pueden mezclar orígenes de datos dentro de una organización, para proveer mayor flexibilidad a la hora de realizar analítica.

Otra funcionalidad clave de la plataforma es que permite generar informes detallados. Esto permite que los agentes puedan evaluar la actividad en su aplicación en determinados intervalos de tiempo. Pudiendo notar dónde están los

mayores puntos de caída, qué secciones les resultan de mayor interés a los usuarios, entre otros.

Uso de usuarios administradores

Los usuarios administradores cuentan con toda la funcionalidad ofrecida a los usuarios de marketing, pero además cuentan con acceso al panel de control de permisos. De esta forma, pueden gestionar que los distintos agentes de marketing no tengan acceso a ciertos datos y/o reportes y pueden dar de baja cuentas de los agentes de marketing.

Mobile Developers & Web Developers

Los desarrolladores mobile y web, pueden subir eventos que registren la actividad de los usuarios cliente en sus aplicaciones, y estos datos son luego consultados por los agentes de marketing para obtener reportes analíticos.

4. Atributos de calidad

4.1. Scalability

- Es fundamental que el sistema pueda escalar horizontalmente para cubrir picos de demanda sin degradación de la performance en caso de que muchas aplicaciones clientes estén registrando datos analíticos de sus usuarios. A su vez, es importante que la arquitectura sea stateless, de forma que se pueda mantener la performance a medida que se levantan instancias a demanda.
- Es necesario poder almacenar una cantidad creciente de datos analíticos de las aplicaciones clientes y de formatos tanto estructurados como no estructurados.

4.2. Availability

- Por necesidad de negocio, es necesario que el sistema de registros analíticos esté preparado para recibir datos y mostrar reportes 24/7. De tal forma que se pueda monitorear en todo momento el volumen de usuarios que usan una aplicación en los distintos horarios, e información pertinente acerca de los mismos.

4.3. Customizability

- Dada su función clave en Business Intelligence y la necesidad de tomar decisiones basadas en grandes volúmenes de datos no estructurados, es fundamental permitir al usuario personalizar de diversas maneras las visualizaciones y los filtros aplicados en el dashboard.
- Se debe poder generar reportes (instantáneos o periódicos) a partir de las distintas campañas de analíticas, con distintas visualizaciones de los datos acumulados. Esto tiene que ofrecerse a través de una interfaz con las distintas opciones de gráficos, segmentos de datos y las posibles opciones de tags/demografías/ubicaciones.

4.4. Performance

- Es necesario que el tiempo de carga de los reportes sea relativamente rápido, si bien no se ofrece un servicio de reporting real time, se busca que varios gráficos y estadísticas estén basadas en datos near-real-time, que tengan demoras en el orden de segundos, es fundamental que el refresh rate no sea muy elevado.
- Es necesario que el tiempo de respuesta del endpoint de tracking sea

lo más corto posible, para que las analíticas no tengan un impacto en performance de páginas actuales.

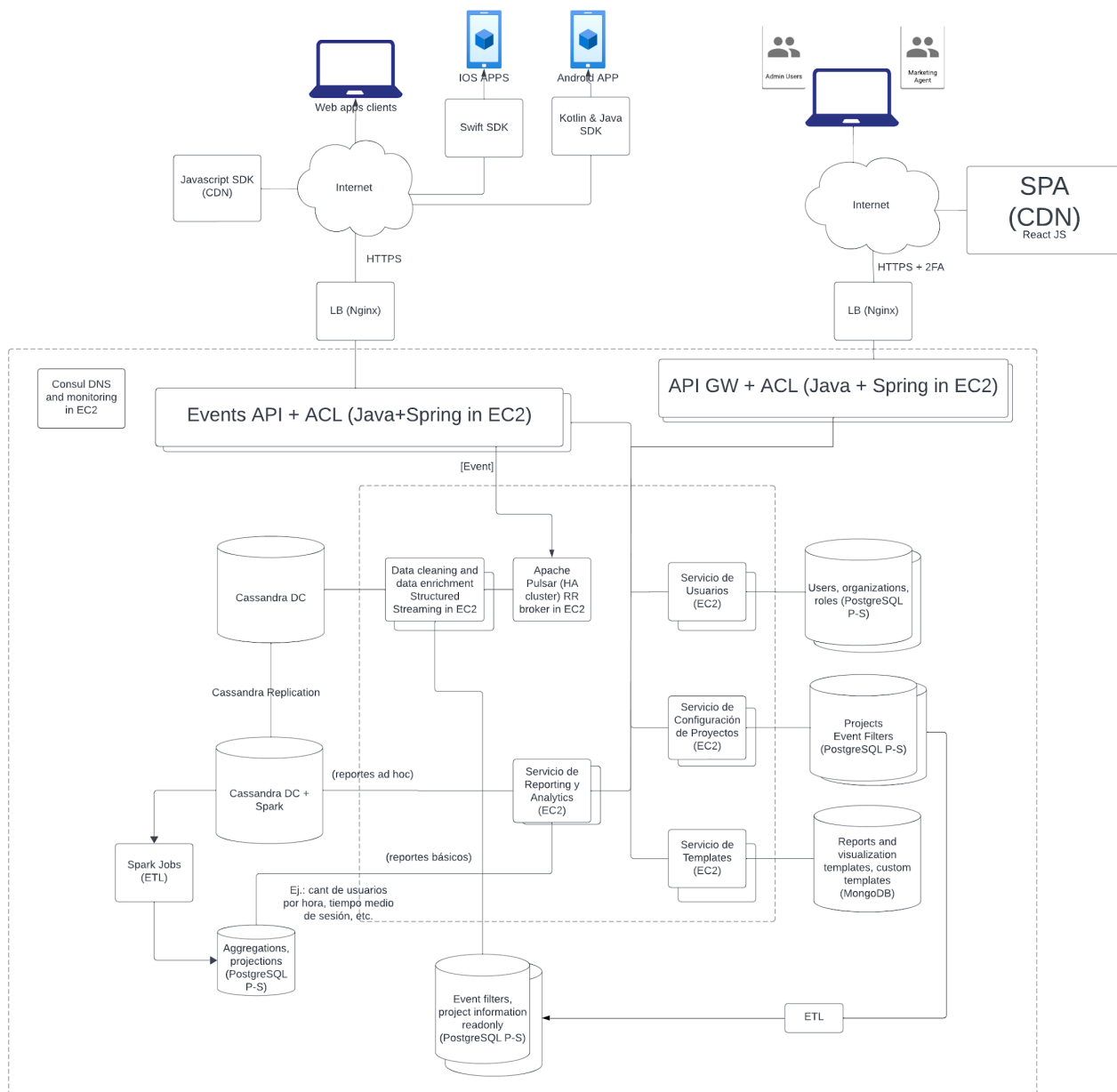
4.5. Security

- Es necesario que agentes de marketing de las diversas agencias no puedan ver datos confidenciales de organizaciones de las que no forman parte. Además es importante que terceros no autorizados no tengan acceso a la plataforma y puedan ver datos que son confidenciales para los agentes de marketing por medio de robos de claves, ingeniería social, ataques cibernéticos, etc.

4.6. Usability

- Debido a que el sistema será empleado principalmente por personas del área de marketing (no técnicas), el mismo debe ser de utilización sencilla.
- Los reportes muy posiblemente sean utilizados durante presentaciones, por lo tanto los mismos deben ser claros, descriptivos, intuitivos e informativos.

5. Vista Física



6. Arquitectura Propuesta

Un sistema de analítica como Google Analytics tiene un gran énfasis en escalabilidad, escritura rápida y un volumen grande de datos. Esto es debido a que es User-Generated data obtenida de interacciones múltiples de una página web o aplicación mobile. Vamos a tener que crear una arquitectura que tenga un gran throughput de escritura sin impacto en la lectura de tales datos, disponibilidad debido a la necesidad de cumplir con un SLA al ser un servicio que puede afectar flujos de negocio, y que tenga una rapidez aceptable a la hora de realizar trabajos de analítica como proyecciones, predicciones y visualizaciones.

Debido a la naturaleza del problema y los múltiples ambientes en el cual se pueden obtener los eventos (un evento se define como una acción que emite el usuario al interactuar con una página web con analítica), se decide crear 2 API's. Un API específico para almacenar y verificar los eventos de los distintos proyectos (un proyecto se define como una campaña específica de analítica, que generalmente se refiere a la analítica de una aplicación web o mobile), y un API Gateway que sirve como punto de entrada para la generación de reportes y datos analíticos. Ambos tienen distintos niveles de control de acceso, debido a que la API Gateway tiene que realizar verificaciones de roles y usuarios mientras que el API de eventos sólo verifica los eventos y su measurementId (identificador de cada proyecto). Ambos APIs están creados con Spring Java y usan EC2 autoscaling. Se realiza Load Balancing a los gateways mediante dos load balancers NGINX levantados con EC2¹.

Ambos servicios de APIs se encargan de hacer ruteo e interfaz de un sistema de microservicios. Se eligió una arquitectura de microservicios debido a los distintos niveles de carga que tiene el sistema y la necesidad de poder escalar ante picos de demanda. Para realizar esto utilizamos el servicio de containers EC2 de AWS, que permite realizar autoscaling a demanda. Hay 2 tipos de contenedores. Hay microservicios que corren contenedores de Java Spring. La decisión de usar Java Spring es debido a la robustez del framework como application framework y contenedor de Dependency Injection. Por otro lado, hay otros servicios que corren Apache Spark con Scala, debido a la necesidad de hacer streaming de datos, mantener high throughput de escritura y para aprovechar posibles mecanismos de procesamiento presentes en el environment de Apache Spark. Se elige Scala debido a que es un lenguaje basado en la JVM con performance y debido a que es el lenguaje con mayor soporte para el ecosistema Spark.

El listado de microservicios y conectores son las siguientes:

1. Microservicio de usuarios y organizaciones. Spring Java.
2. Microservicio de proyectos. Spring Java.
3. Microservicio de templates. Spring Java.
4. Pulsar cluster message broker.
5. Pipeline de data cleaning. Apache Spark Structured Streaming Scala.
6. Microservicio de reportes y analíticas. Spring Java.

¹ [Amazon EC2 Features - Amazon Web Services](#)

7. ETL de agregaciones y analítica. Spark Jobs.

La decisión de utilizar AWS vs On Premises se tomó considerando la necesidad de escalar. Se espera que el tráfico venga de distintas regiones en el mundo, y se espera que este se incremente a medida que más plataformas y servicios adoptan nuestro servicio de analítica. Gracias al servicio de réplicas y autoscaling de EC2, podemos incrementar la potencia de cómputo horizontal y verticalmente, lo cual nos permite mantener un buen performance y velocidad de respuesta. El uso de AWS también nos abre a la oportunidad de levantar el servicio con soporte multi región, teniendo la posibilidad de disminuir la latencia acercando los usuarios finales a las réplicas de región más cercana. La persistencia se replica entre regiones logrando consistencia eventual.

Para la decisión de cloud providers, teníamos las opciones de AWS, GCP y Azure. AWS tiene una gran ventaja en cantidades de data centers, Azure se especializa en una mayor cantidad de regiones y GCP se concentra en optimización de network y baja latencia. Debido a que no tenemos operaciones network-heavy y nos interesa más cercanía física, decidimos ir por AWS. ²AWS también tiene la ventaja de servicios fáciles de utilizar, como su Elastic Compute Cloud, S3 storage y CDN. Priorizamos en disminuir nuestra dependencia en AWS lo más posible, para disminuir la posibilidad de tener vendor lock-in y poder tener más flexibilidad a la hora de tomar decisiones de cloud provider o hosting.

6.1. Arquitectura de microservicios

6.1.1. Load Balancing y autoscaling

Internamente, todos los microservicios descubren a otros con resolución de DNS interno. Las API's también son descubiertas por los load balancers externos mediante este mecanismo. El servicio que utilizan internamente es Consul³, una librería open-source para el descubrimiento de microservicios que ayuda en la resolución de DNS. Cuando una instancia de un microservicio se levanta, hace un request HTTP a la API de Consul, agregando su IP para un determinado hostname. Consul activamente monitorea las distintas IP's para un mismo hostname, realizando health-checks. Si un servicio se llega a caer o fallar, Consul lo detectaría y quitaría su IP para ese hostname.⁴

En cuanto a la resolución del nombre de un hostname, Consul responde con una de las IP's internas que tiene para ese hostname, utilizando la técnica de round robin. De esta manera, la responsabilidad de descubrir microservicios se hace a nivel client-side (es decir, el microservicio que inicia el pedido es responsable de

² [Comparing AWS, Azure, GCP \(digitalocean.com\)](https://digitalocean.com/comparing-aws-azure-gcp)

³ [Discover Services with Consul](#)

⁴ [Introduction to HashiCorp Consul with Armon Dadgar](#)

descubrir a quien se lo hace). Además, este mecanismo DNS es interno, por lo que no está expuesto a internet.

Gracias a Consul, podemos aprovechar la funcionalidad de AWS de autoscaling de EC2. A medida que disminuye la salud de los microservicios (en caso de aumento de demanda), configuramos para que aquellos microservicios puedan expandirse a medida. Para que los load balancers y los propios microservicios puedan hacer discovery de estas nuevas instancias, utilizamos Consul.

Para los microservicios de mayor uso (Pipeline data cleaning, Events API) estos empiezan con una mayor cantidad de réplicas (8) y tienen autoscaling. Para microservicios de menor load (el resto), establecemos una cantidad de réplicas estáticas y sin autoscaling.

6.1.2. Microservicio de usuarios, organizaciones y roles

Hecho en Java y Spring, se encarga del manejo de usuarios, las organizaciones y los roles dentro de la organización. Decidimos crear un sistema de organizaciones debido a que tiene que poder reflejar estructuras empresariales, ya que estos tienden a tener el caudal de interacciones necesario para realizar recolección y visualización de analítica.

Este sistema tiene una jerarquía top down, en el cual un administrador de la organización puede asignar roles y permisos a subordinados de la organización. Los roles son conjuntos de permisos y los permisos son la lectura, escritura, y creación de nuevas visualizaciones o templates. Esto es necesario debido a la posible sensibilidad de los reportes, y la necesidad de diferenciar entre distintos usuarios de analítica dentro de la empresa. Los permisos son por visualización o por conjunto de visualizaciones. El dueño de la visualización es quien tiene el permiso de otorgamiento de permisos sobre la visualización, pero quienes otorgan los permisos también tienen la capacidad de ver el conjunto de visualizaciones de sus hijos. Para compartir visualizaciones, se otorga el permiso de lectura a la visualización. Esta lógica también se aplica sobre los permisos de proyectos, que posteriormente vamos a mencionar en el servicio de reporting y analítica.

Se utiliza JUnit ⁵ con Mockito⁶ para asegurar un testing completo.

6.1.3. Microservicio de configuración de proyectos

La creación de proyectos permite a las organizaciones separar lógicamente distintos orígenes de datos. Esto permite tener una mayor especificación a la hora de analizar las visitas de una página web, la aplicación iOS o la de Android. Cada proyecto tiene un measurement id, que es lo que se utiliza el Events API para saber qué eventos son válidos o no y cómo identificar el proyecto. Además tiene un sistema de filtros sobre un flujo de un measurement id, que permite aplicar lógica

⁵ [JUnit 5](#)

⁶ [Mockito](#)

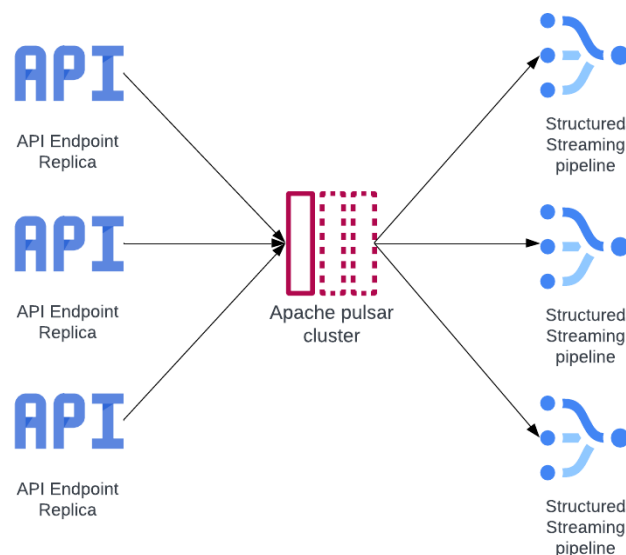
simple ante un evento (ejemplo, filtrar por URL para prevenir eventos maliciosos). Estos son escritos a una base de datos PostgreSQL, con replicación readonly para ser utilizado por el pipeline de eventos. Estos datos son replicados mediante un proceso ETL implementado en Java, el cual corre cada 10 minutos.

6.1.4. Microservicio de templates

Este microservicio está hecho en Java y Spring y se encarga de gestionar los templates tanto predefinidos como personalizados, creados por los agentes de marketing. Ver sección 6.9. para más información acerca de Templates.

6.1.5. Pulsar Worker Queue Cluster

Después de validar el evento, hay que realizar un trabajo de limpieza, filtrado y enriquecimiento de datos. Debido a que no podemos asegurar la velocidad de tales operaciones y queremos que la creación del evento se realice de la forma más rápida posible, decidimos que el Event API se encargue de armar el mensaje y publicarlo a un tópico en Pulsar (ver sección 6.6). Este tópico después es consumido por un conjunto de workers de Apache Spark, y Pulsar se encarga de hacer routing de eventos entre workers disponibles. Este mismo corre en cluster para permitir mínima caída de disponibilidad.



6.1.6. Data cleaning y Data enrichment pipeline.

Para mantener un alto throughput de escritura a la base de datos, se crean workers usando Spark Streaming (ahora llamado Structured Streaming⁷). Este es un framework de Big Data que nos permite especificar operaciones secuenciales sobre conjuntos de orígenes de datos de flujo constante. En específico, vamos a usar un origen y un destino de datos. El primero es el origen de datos de Apache Pulsar, especificando un read stream que lee del cluster de Apache Pulsar. A medida que van llegando los push events de Pulsar, Structured streaming recibe los eventos y los procesa. El segundo es el destino de datos de Cassandra, para almacenar los eventos con muy baja latencia.

En el proceso, el pipeline es el encargado de filtrar eventos que el usuario especificó (definidos en el microservicio 6.1.3). También se encarga de armar la escritura de Cassandra con sus correspondientes columnas.

Este pipeline está escrito en Scala, debido al ser el lenguaje predeterminado de Apache Spark y ser performante al estar ejecutado sobre el JVM.

6.1.7. Microservicio de reporting y analytics.

El microservicio de reporting y analytics es el encargado de otorgar la información agregada y filtrada de los distintos eventos almacenados de los distintos proyectos. Este tiene 2 métodos de recuperación de datos. Para datos near-real time, aprovecha workers Spark que ejecutan queries de agregación, predicción y proyección sobre réplicas de nodos de Cassandra (ver sección 6.3) de forma distribuida. Para datos históricos y simples, como agregaciones por tiempo, por usuarios, por eventos y otras agregaciones que pueden ser precomputadas, se aprovecha la base de datos PostgreSQL OLAP que contiene tal información. La SQL OLAP es actualizada por un Spark ETL (Spark Jobs, ver sección 6.5) que corre regularmente (una vez cada hora) sobre los datos de Cassandra. Esto nos permite tener diferentes niveles de especificación para los datos y visualizaciones. Aquellos que requieren de mayor procesamiento, mayor tiempo de cómputo, visualizaciones complejas o mayor fidelidad a los datos actuales, como reportes ad hoc, podemos aprovechar los workers Spark. Aquellos que tienen mayor uso regular, como dashboards o reportes históricos, pueden aprovechar las agregaciones existentes. Esto nos permite mantener una buena performance sobre los casos de usos cotidianos y mantener la potencia de Spark para un uso específico y analista.

6.2. Load balancers externos

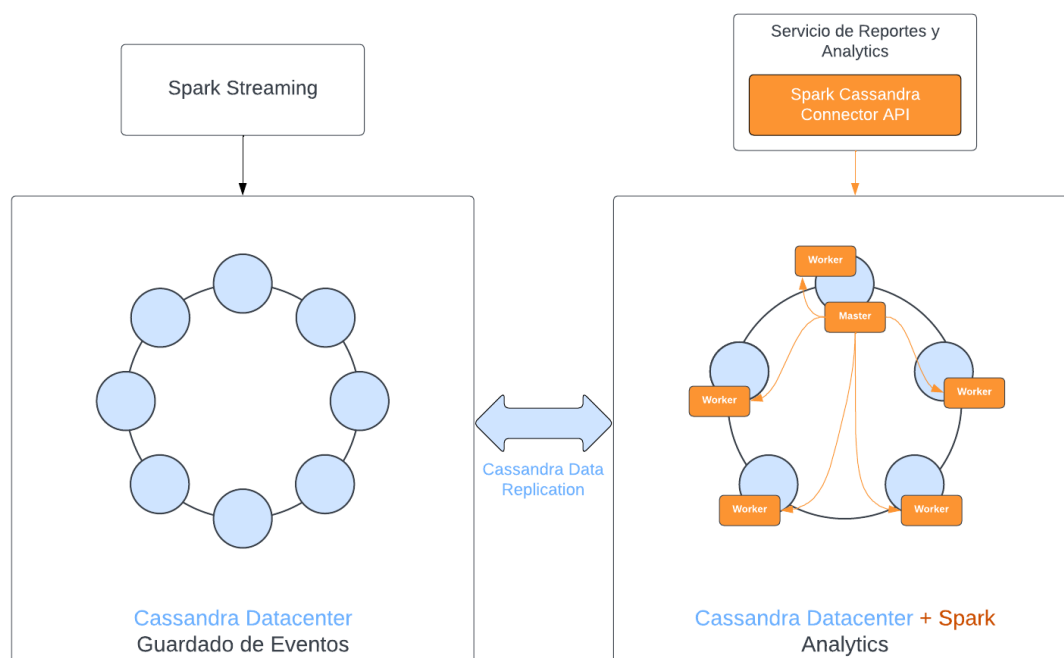
Teniendo en cuenta la necesidad de brindar disponibilidad, se tienen dos load balancers distintos, uno para la API de eventos y otro para la API que utilizan los agentes de marketing. Esto permite por un lado que la caída de un load balancer

⁷ [Structured Streaming Programming Guide - Apache Spark](#)

no inhabilita en su totalidad al sistema ya que si se llegase a caer el load balancer de la API de eventos, no se podrían registrar nuevos eventos pero los agentes de marketing podrían acceder a datos analíticos pues el otro load balancer está disponible. También es posible que ocurra la situación inversa.

6.3. Cassandra con Apache Spark

Teniendo en cuenta la necesidad de una alta disponibilidad y poder generar reportes en near real-time, utilizamos Cassandra con Spark. La recolección de datos se guarda directamente en un Data Center de Cassandra cuyos nodos no cuentan con Apache Spark, Cassandra asegura alta disponibilidad (es AP en el teorema CAP). La generación de reportes no se lleva a cabo sobre esos mismos nodos, Cassandra replicará toda su información relevante en un Data Center con nodos que sí cuentan con Spark. Normalmente para un traslado de datos de este tipo, se hace mediante un ETL de una base de datos a otra, el problema con esta manera de hacerlo yace en la lentitud del proceso. Utilizando el sistema de replicación de Cassandra (el cual es una de las features principales de la base de datos debido a su velocidad), obtenemos la data relevante sin tener que esperar a un proceso Batch (el ETL). Apache Spark se encarga de hacer MapReduce y todas las agregaciones necesarias para generar los Reportes personalizables/configurables.



En el servicio de Reportes y Analytics se utiliza un controlador de Spark (Cassandra-Spark Connector API)⁸, que permite hacer consultas a la base de datos

⁸ [Getting started with the Spark Cassandra Connector Java API | DSE 6.0 Dev guide](#)

Cassandra a través de una instancia de la clase SparkContext que se encarga de comunicarse con el Spark Master. El Spark Master ubica los Spark Workers en cada nodo de Cassandra y les pide que reserven espacio para Executors que realicen el trabajo indicado por el controlador de Spark. El controlador de Spark les enviará la tarea a realizar a los Workers, realizando una operación de MapReduce según la métrica necesaria a calcular para la generación del reporte.

En cuanto al deploy de Cassandra podíamos elegir utilizar Instance Store o Amazon EBS, comparando ambos sistemas, terminamos prefiriendo Amazon EBS debido a la alta cantidad de nodos en el cluster de Cassandra para soportar la inmensa cantidad de datos que va a estar ingresando.⁹

6.4. Estructura en Cassandra

La elección de Cassandra para guardar los eventos se basó en la alta disponibilidad (AP en teorema CAP) y capacidad de escalar de forma horizontal por su arquitectura en cluster de anillo. Esto último provee una escritura muy rápida pero no tan consistente, lo cual no es algo que se priorice para este sistema. La combinación de estas propiedades hace de Cassandra una elección óptima para este tipo de sistemas. Se miraron alternativas similares como HBase, pero la misma presenta una escritura más lenta a cambio de mayor consistencia. Se decidió descartar esta opción para priorizar velocidad sobre consistencia.

En Cassandra, se persisten todos los eventos en una misma familia de columnas. Entre las columnas utilizadas, podemos encontrar:

- eventId: identifica a un evento, sólo con el fin hacer única la Primary Key
- measurementId: identifica a cada proyecto.
- eventType: tipo de evento registrado, pudiendo ser tanto de los predefinidos como los personalizados
- page: ruta del sitio donde ocurrió el evento
- timeStamp: momento en el que ocurrió el evento
- entre otras columnas con información y metadatos del evento como: sessionId, label, value, location, language, viewport, platform, browser, OS, age, gender, etc.

Para esta familia de columnas, se utiliza:

- measurementId como **Partition Key**, de esta manera los eventos relacionados estarán en el mismo nodo.
- (timeStamp, eventId) como **Clustering Key**, de esta forma los eventos están ordenados por el tiempo en que ocurrieron, siendo los más recientes los más relevantes. Se incluye el eventId para que la **Primary Key**, conformada por (measurementId + (timeStamp, eventId)) sea única para cada fila de la familia de columnas.

⁹ [Best Practices for Running Apache Cassandra on Amazon EC2 | AWS Big Data Blog](#)

6.5. Spark Jobs

Spark jobs es la herramienta ETL que brinda el Framework Spark, el mismo es utilizado para generar agregaciones y proyecciones de los datos más comunes para la generación de reportes como pueden ser la cantidad de usuarios en determinada franja horaria o la cantidad de tiempo promedio. Esto hace que la mayoría de los reportes que se generen lo hagan de manera veloz ya que los datos necesarios estarán disponibles antes de que el usuario haga el pedido. El job corre en una instancia de EC2 y está hecho en Scala para aprovechar las funcionalidades del Framework Spark. Este ETL posteriormente guarda los resultados en una base de datos PostgreSQL OLAP.

6.6. Apache Pulsar

El trabajo de Apache Pulsar es de distribuir los eventos que llegan al Events API Gateway. La arquitectura de Pulsar corre en cluster, utilizando 1 o más brokers para realizar load balancing entre diferentes instancias de Pulsar, persistiendo los mensajes en un cluster de BookKeeper en caso de caída, y usando ZooKeeper para realizar tareas de coordinación entre clusters de Pulsar. Como message broker, este expone un HTTP REST API server para hacer discovery de tópicos y publicación/consumo de mensajes. En particular, aprovechamos su capacidad de realizar load balancing entre consumidores un tópico, para mantener a los Spark Streaming Workers a máxima ocupación. Lo configuramos con `receiver_queue_size = 0` para que cada worker tenga que terminar procesando su evento para obtener el otro. Con esto realizamos load balancing óptimo y máximo throughput de procesamiento.

Las opciones que consideramos para message brokers son Apache Kafka y RabbitMQ. Kafka no cumple muy bien cómo work broker, ya que no distribuye mensajes y mantiene mensajes en memoria hasta un cierto tiempo, en caso de una suscripción tardía.

RabbitMQ es muy apto para esta tarea, pero al comparar el throughput vimos que Pulsar es mucho más performante llegando a los 580 MB/s de throughput al mandar datos a los brokers.¹⁰

6.7. Events API

Como capa de entrada para la obtención de datos de los usuarios, las aplicaciones clientes, sean web o móviles, hacen requests HTTPS a una API stateless para enviar datos, esta misma está implementada en Java utilizando Spring Framework. Dicha API, se encarga de asegurar la validez del `measurementId`. Luego, deja los eventos recuperados en el cluster de Apache Pulsar

¹⁰ [Comparing Apache Pulsar vs. Apache Kafka | 2022 Benchmark Report](#)

para que sean publicados y consumidos luego. Los endpoints que expone esta api son los siguientes:

1- GET /track

Este endpoint permite trackeo mediante web-beacons (ver sección 6.8.2). Brindando acceso a una imagen dummy transparente de 1x1 pixels, indistinguible para el ojo humano.

2-POST /track

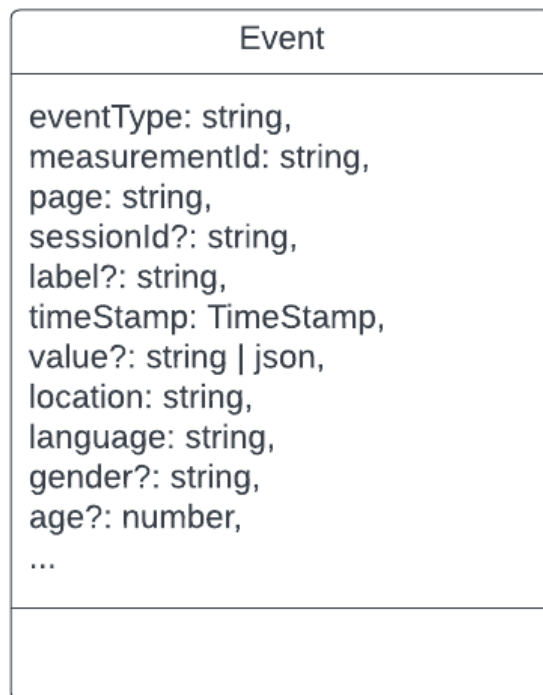
Permite pasar en el body del request un JSON que tenga información relevante de los eventos, para poder hacer el trackeo de las analíticas que sean necesarias.

Para el caso de Requests de tipo POST, se envía información y metadatos del evento de la siguiente manera:

```
POST HTTP Request:
Headers → se obtiene: language, platform, browser, hostname, location, viewport, OS, etc.
Body → (json) {
  eventType: string,           //click, conexión, swipe, compra, etc.
  measurementId: string,      //identifica a cada proyecto
  sessionId?: string,         //el desarrollador lo puede incluir para identificar la sesión en la que ocurren
  label?: string,             //el desarrollador puede agregar información adicional
  value?: string | json,      //que desee con los campos label y value
  gender?: string,            //si posee la información de los usuarios,
  age?: number,               //puede agregar información personal del usuario
}
```

Para el caso de Requests de tipo GET con Web Beacons, se envían los parámetros del body como QueryParams.

La API realiza las transformaciones necesarias para enviarle al cluster de Pulsar una entidad de este tipo:



6.8. SDKs

Tanto los SDK de Javascript para Web como Kotlin y Swift para Mobile permiten el envío de eventos ocurridos en la aplicación a la API de eventos a través de requests HTTPS. Se indica el tipo de evento que puede ser tanto uno de los eventos básicos (click, conexión, swipes, drags, etc.) como eventos personalizados que defina el usuario.

6.8.1. Kotlin SDK, Java SDK y Swift SDK

Hosteando el SDK en Maven le permite a usuarios tanto de Maven como de Gradle utilizar nuestra dependencia. Mediante la misma se pueden utilizar distintas clases, en las cuales el desarrollador puede guardar distintos tipos de eventos en nuestro sistema y subsecuentemente visualizar desde nuestro SPA los reportes generados.

Dichas clases le otorgan al desarrollador manejo sobre cuáles son los eventos (scrolling, swiping, etc) considerados pertinentes para los reportes.

En el caso del SDK para desarrolladores de Swift, el SDK está hosteado en SPM (Swift Package Manager) y le ofrece las mismas herramientas que el SDK para desarrolladores de Kotlin y Java.

6.8.2 Javascript SDK

El javascript SDK se expone únicamente un archivo js mediante la CDN, para esto utilizamos Amazon CloudFront. Los desarrolladores web pueden importarlo a su sitio con un tag `<script>` en el header HTML al estilo:

```
<script src="https://path.../index.js"></script>
```

Este script define listeners para acciones del usuario en la web (click, cambio de página, etc.), para así poder reportar estos eventos en formato JSON a través de un POST HTTP a la API de eventos.

Para eventos de carga uno puede usar web beacons, que realizan llamadas tipo GET al API mediante la carga de un componente renderizable (``, `<iframe>`). Esto obliga al browser cargarlo y los datos se codifican como query params. Uno simplemente copia el web beacon en la página que quiere trackear y automáticamente maneja eventos de carga de la página.

Además, cuenta con la definición de métodos para que el desarrollador pueda enviar eventos personalizados a la API en el momento que desee pertinente.

6.9. Templates.

A partir de los eventos guardados en la BD Cassandra, a través de Spark se puede calcular métricas (*Metrics*) como por ejemplo: cantidad de clicks, cantidad de conexiones, tiempo medio de sesión, etc. Para calcular estas métricas, se deberán realizar operaciones de MapReduce distintas dependiendo de lo que se requiera. Debido a que los eventos pueden ser personalizados por el cliente, las métricas que se ofrecen para calcular a partir de estos eventos personalizados son preestablecidas: cantidad total de un evento, frecuencia de un evento por intervalo de tiempo, tiempo medio entre eventos, etc. Este trabajo se realiza en el servicio de reporting y analytics.

Por otro lado, los eventos almacenados tienen atributos asociados (*Dimensions*) con metadatos del evento, por ejemplo: idioma del usuario, plataforma, edad del usuario, ubicación de la conexión, etc.

A partir de estos *Metrics* y *Dimensions* se definen los gráficos como *Templates*.

Template
<div>+ graphType: bar line piechart ... + interval_start?: Timestamp + interval_end?: Timestamp + granularity?: hour day month + groupBy?: Dimension + ... + first_axis: Metric + seconds_axis?: Metric + filters: Filter[]</div>

Estos son objetos que contienen el tipo de reporte (tabla, gráfico de líneas, gráfico de barras, gráfico de torta, etc.), el intervalo y granularidad del tiempo en el caso que sea un gráfico en el tiempo, qué *Metric* es representada en cada eje del gráfico y más propiedades que dependerán de cada tipo de gráfico. Además, permite guardar una *Dimension* como criterio de agrupación del gráfico (por ejemplo para crear un gráfico de línea en el tiempo con múltiples líneas, una por cada grupo). Finalmente, permite agregar filtros a los datos a incluir tanto según el valor de una *Dimension* como por el de una *Metric*.

Existen algunos templates básicos predefinidos de los gráficos que se muestran en el dashboard principal de la interfaz web, como por ejemplo conexiones por hora, ubicación de las conexiones del último mes, o tiempo medio de sesión.

Por otro lado, se ofrece una interfaz intuitiva para los agentes de marketing que deseen generar templates propios de gráficos y reportes personalizados. En esta pueden definir las propiedades del template a través de un menú y ver la previsualización del gráfico que están generado con ese template. Finalmente, pueden guardar estos templates personalizados para consultarlos más adelante.

Tanto los templates predefinidos como los personalizados se almacenan como documentos en una Base de Datos MongoDB, debido a la flexibilidad que se necesita para la estructura del Template según el tipo de reporte que represente.

Los templates terminan siendo instrucciones de renderizado para frontend. El SPA trae el template a renderizar, sea predefinido o personalizado, y puede insertar o ya puede venir con una referencia (un *measurementId*). Con tal referencia y los filtros y/o operaciones de agrupamiento definidos en el template uno envía esta solicitud para ser procesado en el microservicio de analítica y devuelve la sucesión de datos necesarios para popular los reportes y/o visualizaciones.

6.10. API Gateway

Como capa de entrada a los microservicios, los agentes de marketing y administradores utilizan una API GW que expone endpoints para obtener información de los distintos datos analíticos que se requieran consultar.

Dentro de esta API, se encuentra una ACL para controlar el acceso a los distintos recursos para los agentes de marketing. El manejo de sesión se hace mediante JWT. Esta API es REST, por lo tanto no mantiene estado.

6.11. SPA

La SPA brindada a los agentes de marketing está creada con React (v18.0), debido a la gran popularidad del framework, su comunidad activa y la compatibilidad con una gran cantidad de navegadores. Adicionalmente, usa la librería Chart.js para mostrar los gráficos que los agentes de marketing precisen

para obtener la información que necesitan. Este va a ofrecer una UI con templates de dashboards pre-definidos, y un menú para configurar filtros y segmentación de los datos. Con este menú uno puede crear nuevas visualizaciones y reportes, también realizar proyecciones de tendencias de los eventos y visitas. La idea es que con la creación de visualizaciones uno puede personalizar fuertemente cómo realizar los trabajos de analítica.

El SPA también tiene toda la configuración de un proyecto y una organización, permitiendo crear la estructura, roles y permisos de la organización para que refleje el orden de visibilidad que se le quiere otorgar a cada flujo de datos existentes.

Para asegurar la usabilidad del SPA procedemos a realizar entrevistas a agentes de marketing con preguntas relacionadas a qué hacen en su día a día y cómo utilizan o quisieran utilizar datos de sus usuarios. Además, se hacen pruebas con prototipos de distintos niveles de fidelidad, pidiéndole a los potenciales usuarios que realicen distintas tareas en el sistema para analizar qué tan intuitivos son los workflows del sistema. Los potenciales usuarios fueron elegidos basándonos en los modelos de persona desarrollados.

El SPA está hosteado en un CDN. En particular, decidimos utilizar S3 y Amazon Cloudfront debido a su cobertura y availability zones.

Se utiliza Jest para el testing de la SPA. Nos aseguramos de una correcta internacionalización y localización mediante el uso de React-intl.

6.12. PostgreSQL

Para las operaciones transaccionales con schema fijo decidimos utilizar PostgreSQL. Esto es debido a que es una base de datos open source, comunidad abierta, de amplio soporte y con muy buen performance a la hora de realizar queries. Se levantan múltiples instancias EC2 hosteando PostgreSQL, todas con redundancia Primary-Secondary para asegurar la disponibilidad para transacciones críticas y permitir escalar las lecturas.

Hay una base de datos por cada microservicios, para mantener mínimo acoplamiento entre microservicios. Para el pipeline de data cleaning y data enrichment es necesario información del proyecto y los filtros de eventos presentes en tal, por lo tanto se mantiene una réplica readonly mediante un ETL que se corre cada 10 minutos. Esto disminuye el load existente en la base de datos principal de Proyectos, y permite minimizar el acoplamiento entre estos dos servicios. Para mantener durabilidad en la instancia de EC2, se utiliza un Elastic Block Store como storage.

7. Puntos críticos del sistema

7.1. Recolección de Datos

Como pieza principal del sistema se encuentra la recolección de datos a través de eventos, enviados tanto por páginas web como por aplicaciones móviles. Por lo tanto, es imperativo que se pueda recibir y guardar la mayor cantidad de eventos posibles. Al tener una combinación de tecnologías de alta disponibilidad, tolerancia a fallos y escalabilidad tales como Apache Pulsar, Structured Streaming y Apache Cassandra como Base de Datos, se garantiza una pérdida mínima de disponibilidad.

7.2. Procesado de Datos para Reportes

Como segunda pieza se encuentra el procesado de los datos recolectados a través de los eventos. Lo crítico se puede ver en la magnitud y variedad de los datos que tendrán la mayoría de los proyectos monitoreados. Por ello, se decide hacer uso de Spark en conjunto con Cassandra, con un conector especial de DataStax¹¹, para poder manejar una gran cantidad de datos junto con la posibilidad de realizar agregaciones complejas. Con esto, es posible realizar el procesamiento necesario para la realización de los reportes tanto personalizados como predefinidos. Para estos últimos en especial, se hace uso de Spark Jobs (con frecuencia de una vez por hora) para realizar agregaciones y proyecciones, de forma tal que el uso de estas métricas en los reportes predefinidos se vea optimizado.

7.3. Generación y Visualización de Reportes

Como tercera pieza está la generación y visualización de los reportes, la cual debe ser performante, clara y personalizable por cada usuario. Primero y principal, la performance se puede garantizar a través del uso de Spark para el procesamiento de datos, tanto para los reportes personalizados como predefinidos. En segundo lugar, los reportes deben ser claros por lo que se provee una interfaz intuitiva imitando las aplicaciones más usadas por los usuarios, tales como Tableau. Finalmente, se provee una interfaz gráfica para poder elegir las opciones de cada reporte, qué datos incluir, filtros en base a métricas calculadas, segmentos de tiempo, etc.

7.4. Acceso a los microservicios

Como última pieza, pero no menos importante, es atender a una gran cantidad de pedidos sin afectar la performance. Por lo que, se decidió implementar

¹¹ [datastax/spark-cassandra-connector: DataStax Connector for Apache Spark to Apache Cassandra \(github.com\)](https://github.com/datastax/spark-cassandra-connector)

el patrón API Gateway para los pedidos de los usuarios de la SPA y una API para la recepción de eventos. Cabe destacar que, ambas entradas a los microservicios están replicadas y los pedidos están distribuidos por un Load Balancer por lo que un aumento de los mismos no afecta a la performance. Para discovery entre microservicios, utilizamos Consul que hace el trabajo de DNS para identificar clusters de microservicios mediante hostname. Esto permite al Load Balancer distribuir la carga y que los microservicios puedan descubrirse entre sí.

8. Escenarios

8.1. Scalability

- Escenario: Hay un gran pico de uso en uno de los proyectos y genera una enorme cantidad de eventos para registrar.

Solución: Se hace uso de Apache Pulsar para encolar todos los eventos que llegan de forma tal que no se pierda ningún evento sin procesar. Estos eventos se entregan a Structured Streaming que facilitan el procesamiento gracias a su alto nivel de performance para streams de datos. Finalmente, se usa Cassandra para poder manejar la gran cantidad de escrituras.

- Escenario: Uno de los clientes necesita datos históricos para su análisis y estima que se duplicará la cantidad de eventos que se generan cada 6 meses.

Solución: Se hace uso de Apache Cassandra como base de datos principal de forma tal que se pueda escalar horizontalmente de manera sencilla para manejar los aumentos en la cantidad de eventos que se guardan.

8.2. Availability

- Escenario: Un proyecto tiene tráfico las 24 horas del día y es necesario que se capturen la mayor cantidad de eventos posibles.

Solución: Se utiliza un Load Balancer, varias instancias de la API de Eventos junto con Apache Pulsar, Spark Streaming y Cassandra. Las mencionadas tienen medidas para prevenir fallas que limitarían acceso a los servicios por lo que no se perderá la recepción y guardado de eventos.

- Escenario: Se caen las bases de datos de Users, Projects o EventFilters (PostgreSQL).

Solución: Se realizan esquemas Master-Slave, por lo que en caso de la pérdida de un Master no se pierde el acceso a los datos

- Escenario: Se cae la base de datos de Templates(MongoDB)

Solución: Al ser MongoDB esta trabaja en Cluster y usa esquemas Master-Slave para evitar que la caída del Master signifique la pérdida del acceso a los datos.

- Escenario: Se cae la API Gateway.

Solución: Se tienen varias instancias de la misma junto con un Load Balancer para direccionar los pedidos a las instancias activas.

- Escenario: Se cae un conjunto de nodos de Cassandra

Solución: debido a la arquitectura distribuida y replicada de Cassandra, siempre hay nodos presentes para procesar escrituras por más que otros nodos estén caídos.

- Escenario: Se caen algún microservicio

Solución: cada microservicio tiene réplicas estáticas que permiten seguir funcionando a pesar de algún fallo de sus réplicas.

8.3. Customizability

- Escenario: Se quiere construir un nuevo reporte a partir de un template preexistente.

Solución: El editor de reportes del SPA permite la copia y edición de templates y la conexión con datos de los eventos. Este permite agregar nuevas visualizaciones, obtener nuevas proyecciones y crear reportes personalizados. Estos templates hechos después persisten en las bases de datos de templates personalizadas.

- Escenario: A pocos días de una reunión, el líder de marketing le pide a uno de sus subyugados que genere un reporte en base a métricas poco ortodoxas.

Solución: Mediante la interfaz de creación de reportes, se pueden generar tipos de reportes personalizados con gran facilidad por medio de varios menús de opciones que le ofrecen al agente de marketing combinar los distintos tipos de métricas para alcanzar lo que está buscando.

8.4. Performance

- Escenario: Se decide realizar un reporte usando gran cantidad de datos históricos y se espera que se obtenga una visualización en menos de 5 segundos.

Solución: A través del uso de Spark con varios Workers y Map-Reduce se puede orquestar un filtrado de una gran cantidad de datos en muy poco tiempo, por lo que se puede obtener el reporte de una forma rápida.

- Escenario: Se necesita mostrar en el dashboard los reportes predefinidos en menos de 2 segundos.

Solución: Se realizan agregaciones y proyecciones una vez cada hora con Spark Jobs y se guardan en un PostgreSQL OLAP. Esto permite ser fácilmente extraídos por Spark para realizar los reportes predefinidos cada vez que se necesiten.

- Escenario: Se necesita que la llamada que envía los eventos tenga el menor tiempo de respuesta posible.

Solución: el Events API se encarga de dejarlo en la Workers Queue de Pulsar, que asegura la presencia del mensaje en el sistema pero permite una respuesta más rápida al deferir el procesamiento del evento a un futuro cercano.

8.5. Security

- Escenario: Alguien ajeno a cierta organización obtiene el nombre de usuario y contraseña de un empleado y pretende visualizar datos a los que solamente los miembros de dicha organización tienen acceso.

Solución: El 2-FA le provee una capa extra de protección a dicho usuario y a la organización.

- Escenario: Se planea realizar un ataque de MITM(Man-In-The-Middle) para robar credenciales.

Solución: Se bloquea esta posibilidad usando HTTPS para toda comunicación externa.

- Escenario: Se obtiene el measurementId por lo que se puede colocar el código de JS en otra página, pudiendo engañar al sistema con métricas que no son reales o enviando demasiados eventos para saturar al sistema.

Solución: Se pueden realizar filtros que solo acepten eventos de un hostname específico para que no puedan usar el measurementId en otras paginas y para una cantidad excesiva de pedidos se implementa un rate-limit por IP.

8.6. Usability

- Escenario: Un agente de marketing con poco conocimiento técnico utiliza el sistema por primera vez.

Solución: Los flujos de la aplicación entre pantalla y pantalla son claros y el usuario en todo momento sabe en qué parte se encuentra y cuáles son sus opciones. Esto se logra realizando tests de usabilidad con posibles clientes y agentes de marketing.

- Escenario: Un developer nuevo quiere introducir analytics de una forma fácil y simple en su nuevo proyecto en auge.

Solución: La página de desarrollo está documentada y actualizada y hay snippets de código fáciles de copiar y utilizar en diferentes lenguajes (ej: JavaScript, Kotlin, Swift).

9. Supuestos, riesgos, no riesgos y trade-offs

Supuestos:

- No se requiere que todos los reportes mostrados en la interfaz se actualicen en real-time. Basta con que se muestran near real time.
- Suponemos que los agentes de marketing van a contar con acceso a internet y dispositivos celulares para 2FA.

No Riesgos:

- Si se cae la API GW, hay otras instancias disponibles para recibir requests.
- Si se cae un microservicio en particular, tenemos varias instancias.
- Frente a un ataque DDOS, los LB cuentan con mecanismos de rate limiting para una misma IP. Evitando así que se le niegue el servicio a los demás clientes.
- En caso de ataque como man in the middle, la comunicación ocurre mediante HTTPS, el tráfico resulta encriptado, brinda Seguridad.
- En caso de robo de claves de cuenta con ingeniería social, se cuenta con 2FA, por lo que además se requerirá acceso al código enviado por teléfono celular. Bloqueando así el robo de sesión. Esto aporta seguridad adicional al sistema.
- Bases de datos SQL OLTP se encuentran replicadas en un esquema Master-Slave, por lo tanto, en caso de caída de la base maestra, una esclava puede tomar su lugar.
- La cola de mensajes de Pulsar corre en cluster por lo que la caída de un nodo del cluster no implica la falla total de la cola de mensajes. Esto brinda una alta disponibilidad y tolerancia a fallos.
- Hay varios procesos consumidores de los datos que se obtienen de la cola de mensajes, si falla o se cae uno, hay otros que pueden tomar su lugar.
- Utilizar una arquitectura de microservicios, permite escalar a nivel microservicio las instancias de forma horizontal frente a picos de demanda. Sin necesidad de escalar todo el sistema como en un monolito.
- La cantidad de datos ingresados en el sistema no afecta la performance total debido a ser guardado en Cassandra.
- La arquitectura de todo el sistema, resulta stateless. Por lo tanto, al escalar frente a fuertes picos de demanda o necesidad de crecimiento del sistema, la performance no se degrada como ocurre en una arquitectura stateful que usa sticky sessions.
- Un pico en escritura de Cassandra no afecta el throughput debido a la escritura distribuida en Cassandra.

Riesgos:

- Los load balancers presentan un single point of failure. Si se cae un load balancer, una parte importante del sistema queda inhabilitado, en caso de que sea el que gestiona el tráfico de la API de eventos, no se podrán subir

trackeos, por otro lado, si falla el que gestiona el tráfico para el acceso a la visualización de los reportes analíticos, las estadísticas no serán visibles.

- Si a un agente de marketing le roban las claves y el código enviado a su teléfono celular. Un tercero no autorizado podría obtener acceso al sistema.
- Hay una caída regional en los servicios de EC2 de AWS. Debido a que el sistema está deployado en AWS, dependemos de un tercero en la salud de nuestro sistema en general.
- Surge en popularidad el desactivamiento de Javascript. Para páginas web que funcionan sin javascript pero tienen tracking, esto impediría el tracking de datos mediante el SDK. Las cargas de página usando web beacons siguen funcionando al ser codificados como un elemento no ignorable (img o iframe), pero disminuye la cantidad de datos que uno recupera.

Tradeoffs:

- **Scalability over Performance:** al usar Consul y resolución de DNS, aumenta la latencia en respuesta. Esto consideramos un tradeoff razonable ya que esto nos permite realizar escalado de réplicas ante un aumento de load.
- **Security over Usability:** El uso de 2FA puede impactar negativamente en la experiencia de uso de un usuario, dado que es una tarea tediosa de realizar.
- **Availability over Costs:** Mantener un alto nivel de instancias de distintos microservicios y bases de datos para brindar una alta disponibilidad resulta costoso.
- **Scalability over Costs:** Mantener una arquitectura de microservicios es costoso, debido a que te cobran por tiempo de computación. Si uno incrementa la cantidad de capacidad de cómputo en AWS, puede aumentar significativamente los costos de ejecución.
- **Scalability over TTM:** Una arquitectura de microservicios permite una mejor escalabilidad que una arquitectura de monolito, dado que se puede escalar a nivel microservicio sin necesidad de levantar una nueva instancia entera del sistema. No obstante, la naturaleza distribuida de los microservicios requiere lidiar con consideraciones adicionales que demoran la entrega del producto al sistema (ej: Service discovery, etc...).
- **Performance over Security:** Para mejorar la performance al realizar interacciones con las bases de datos. Se optó por no usar encryption at rest, debido al overhead que implica este mecanismo.

10. Bibliografía


Cassandra y comparación con alternativas:

- [HBase vs Cassandra: Which is The Best NoSQL Database \(appinventiv.com\)](https://appinventiv.com/blog/hbase-vs-cassandra-which-is-the-best-no-sql-database/)
- [Cassandra Partition Key, Composite Key, and Clustering Key | Baeldung](#)
- [Best Practices for Running Apache Cassandra on Amazon EC2 | AWS Big Data Blog](#)

Comparación entre Pulsar y alternativas:

- [Kafka vs. Pulsar vs. RabbitMQ: Performance, Architecture, and Features Compared](#)
- [RabbitMQ frente a Kafka: diferencia entre los sistemas de colas de mensajes - AWS](#)
- [Use Pulsar as a message queue | Apache Pulsar](#)
- [Comparing Apache Pulsar vs. Apache Kafka | 2022 Benchmark Report](#)

Spark-Cassandra Connector API:

-  [Spark and Cassandra: An Amazing Apache Love Story - Patrick McFadi...](#)
- [spark-cassandra-connector/doc/8_streaming.md at master · datastax/spark-cassandra-connector \(github.com\)](#)
- [datastax/spark-cassandra-connector: DataStax Connector for Apache Spark to Apache Cassandra](#)
- [Getting started with the Spark Cassandra Connector Java API | DSE 6.0 Dev guide](#)


Overview Spark y ejemplos:

- [Apache Spark](#)
- [Overview \(Spark 3.5.0 JavaDoc\)](#)
- <https://github.com/apache/spark/tree/master/examples/src/main/java/org/apache/spark/examples>
- [Structured Streaming Programming Guide - Apache Spark](#)

AWS y comparación con alternativas:

- [Amazon EC2 Features - Amazon Web Services](#)
- [Comparing AWS, Azure, GCP \(digitalocean.com\)](#)

Consul:

- [Discover Services with Consul](#)
-  [Introduction to HashiCorp Consul with Armon Dadgar](#)

Testing:

- [Mockito](#)
- [JUnit 5](#)

Miscellaneous:

- [NGINX Rate Limiting](#)
- [react-intl - npm](#)