## 

Documento de Arquitectura

Laboratorio III

*Profesores:*

*Martin Salaberri*

*Mariana Falco*

*Ignacio Berdiñas*

*Matias Cicilia*

*Alumnos:*

*Maximiliano Adaro*

*Katia Cammisa*

*Pedro Gardeliano*

*Joaquín Iannuzzi*

Índice

[**Condicionantes de la arquitectura alcanzados**](#_g873ymoade3f) **3**

[El soporte/mantenimiento debe ser lo más simple posible.](#_msefh0b5s3e3) 3

[Necesidad de poder conectar con distintos servidores.](#_8b2h0au7wasc) 3

[Necesidad de guardar grandes cantidades de datos](#_fox12bkvr81e) 3

[Poder operar sobre grandes cantidades de datos (Machine learning)](#_e9wbojghnbzg) 3

[Soportar grandes procesamientos de datos (Machine learning)](#_l6g026t35dtn) 4

[Mapa con ubicaciones de los inmuebles en venta/alquiler](#_8qjnbeayecmc) 4

[Necesidad de tener una aplicación Web responsive](#_i58sj9ffeh9e) 4

[Facilidad de inicio de sesion de usuarios](#_tqzw3gq7bk2c) 4

[**Condicionantes de la arquitectura (a futuro)**](#_cq5ia76hb46p) **4**

[Necesidad de poder conectar con distintos servidores.](#_6ubf9abl3f1z) 5

[Necesidad de Backups (remotos)](#_bdoz4co8xxak) 5

[Soportar una cantidad grande de información multimedia](#_636wn09dwfrc) 5

[Necesidad de streaming de información multimedia](#_pbl0xwif0o5) 5

[Zero downtime deployments](#_i1enyisxv142) 5

[Deploys dirigidos](#_bmhsavu70isg) 6

[Sitio liviano para dar respuesta a celulares de gama baja](#_wtqpseh9q82x) 6

[Observabilidad](#_8keaw9sfz4yw) 6

[Logging](#_h65t45digj6g) 6

[Monitoreo](#_56gnhm1vw4wc) 6

[Alarmas](#_9mk83iqlvsu2) 6

[Nice to have](#_dx8wzikemcp8) 6

[Optimización de costos](#_4864y9jy0dp8) 6

[Trazabilidad de versiones end-to-end](#_9a895pq3azfg) 6

[Transparencia de ubicación de los servicios](#_wq4qzrrb0ru4) 7

[Proceso de deployment simple](#_nfh6785omvfn) 7

[**Diseño de la Arquitectura**](#_3kral2gks57p) **8**

[Amazon Route 53:](#_4bsfag6b5w6) 8

[Amazon CloudFront:](#_9o57rh8k3bmu) 8

[Amazon S3:](#_3cwsblr5gw5m) 8

[Amazon Elastic Beanstalk:](#_m26ki1jxe7sl) 9

[Amazon Application Load Balancer:](#_dslydwo5mleu) 9

[PostgreSQL:](#_a1e44zhb96g6) 10

[Amazon Relational Database Service:](#_4cr6m7mpba7k) 10

[Aws Lambda:](#_ul0byrbqoe5v) 10

[**Machine Learning:**](#_oygbjpjyoiz2) **11**

[Python:](#_pv5yphglvzo7) 11

[Amazon Elastic Container Registry:](#_ajg3rkkhp8kb) 11

[Docker:](#_8t4n0owpx2co) 11

[**Monitoreo:**](#_5ydac0wop05q) **12**

[Amazon CloudWatch:](#_gz1npq2fcl6q) 12

[**FrontEnd:**](#_oot04pt94tg9) **12**

[Spring:](#_dy49ipzc3nko) 12

[Kotlin:](#_8d1fl45yxbin) 13

[**BackEnd:**](#_v02s9ssdgjkv) **13**

[React:](#_v57szhmklt3v) 13

[TypeScript:](#_jt43tlc9srgy) 14

[**Estructura de Costos**](#_rqbogsglciqb) **15**

[Amazon CloudFront:](#_2ded1lndpm3q) 15

[Amazon CloudWatch:](#_rz9jha4lshm1) 15

[Amazon Elastic Beanstalk:](#_l9fd4jvsltz3) 16

[Auto Scaling Group:](#_41v32zp02ef3) 16

[Amazon Route 53:](#_tmxz5erlzq61) 16

[Amazon Application Load Balancer:](#_xrlp9ubnp1yp) 17

[Amazon Elastic Container Registry:](#_7dux8s7kof6t) 17

[Amazon RDS:](#_ehxl0rl6ux60) 17

[EC2:](#_j2ois4uuvvbl) 17

[EBS:](#_2gwe4r5bp49g) 18

[S3:](#_q7dc8p6xhmxk) 18

[AWS Lambda](#_rvte9d8aq5t9) 18

[Total](#_zhdjg0udb9ek) 19

**Ubicar**

# **Condicionantes de la arquitectura alcanzados**

## El soporte/mantenimiento debe ser lo más simple posible.

* Para asegurar una rápida corrección de errores. Lo ideal sería poder solucionar cualquier bug para tener el sistema corriendo y sin fallas en el menor tiempo posible.

## Necesidad de poder conectar con distintos servidores.

* Recoger información geoespacial de las zonas mediante un servidor público que la provee. Estos datos se actualizan mensualmente por lo que se deberían hacer pedidos a la API en ese plazo.

## Necesidad de guardar grandes cantidades de datos

* Se necesitará acceder a estos datos para proveer la información geoespacial relevante de cada zona disponible en la página.
* Los datos son sobre la residencia y la zona en la que se encuentra. Por ejemplo, cuántos y cuáles edificios de seguridad hay en la zona elegida. Así como también cuenta con información de los servicios de transporte, por ejemplo.
* Para los datos geoespaciales estamos estimando un total de 200 MB en archivos geojson (Basándonos en archivos .geojson disponibles online de IGN e IDERA)
* Para los datos residenciales estamos estimando de 1GB (500.000 propiedades \* 2KB)

## Poder operar sobre grandes cantidades de datos (Machine learning)

* Una función clave de nuestra aplicación es que cuenta con un sistema de notificaciones de oportunidades de inversión. Esto se logra aplicando Machine Learning sobre los datos propios de un inmueble y su información geoespacial para reconocer cuando una propiedad se encuentra a un precio más bajo del usual en esa zona.

## Soportar grandes procesamientos de datos (Machine learning)

* Periódicamente se necesitan entrenar los datos y probarlos para asegurar un buen funcionamiento.

## Mapa con ubicaciones de los inmuebles en venta/alquiler

* Debe incluir un mapa interactivo donde se muestran los diferentes inmuebles disponibles para su compra y venta con sus respectivos precios y características.
* Posibilidad de aplicar diferentes filtros al mapa para obtener resultados según la búsqueda del usuario

## Necesidad de tener una aplicación Web responsive

* Nuestro servicio está basado en una aplicación web, específicamente responsive, para que se pueda usar en todo tipo de dispositivos.

## Facilidad de inicio de sesion de usuarios

* Utilizar plataformas como Firebase que faciliten el registro de usuarios dentro de la aplicación. Como el logueo de usuarios vía su cuenta de Google.

# **Condicionantes de la arquitectura a futuro**

## Necesidad de poder conectar con distintos servidores.

* Conexión a Mercado Pago para efectuar transacciones de pago de comisiones o promoción. (no se llegó a integrar con mercado pago)

## Necesidad de Backups (remotos)

* En el caso de un fail en el servidor, necesitamos tener copias de seguridad para poder restablecer nuestro servicio.
* Se quiere backupear diariamente las publicaciones creadas y los reportes de inspección. Luego se querrían almacenar quincenalmente el resto de los datos de la aplicación.
* Lo ideal sería mantener un sistema automático de back-up semanal, pero también se podría ejecutar uno manualmente en caso de que se desee. Este backup podría estar motivado por el pronto deploy de un feature que pueda generar problemas. Para evitar la pérdida de datos o el mal funcionamiento de la aplicación, lo recomendable sería forzar un backup.
* Con respecto a un plan de ‘disaster recovery’ estamos dispuestos a perder un registro de hasta 24 horas con la idea de tener el sistema funcionando de nuevo en menos de 3 horas.

## Soportar una cantidad grande de información multimedia

* Para cada publicación se va a necesitar guardar varias imágenes y los datos técnicos de cada propiedad.
* Se pueden subir fotos con un máximo de 5MB en total (200 kB x 25 fotos)

## Necesidad de streaming de información multimedia

* Lo que más va a gastar recursos del sistema es la bajada de información multimedia. Necesitamos que este servicio esté optimizado para reducir los mismos.

## Zero downtime deployments

* Necesitamos que en actualizaciones del servicio, no haya interrupciones en el sistema. Que la transición sea transparente y fluida.

## Deploys dirigidos

* Podemos hacer deploys por secciones para hacer pruebas en una parte del mercado antes de deployar el feature en toda la aplicación.

## Observabilidad

### Logging

* Para poder ver en específico si hay fallas que fue lo ocurrido.

### Monitoreo

* Para poder monitorear si los servicios son adecuados y son funcionales.

### Alarmas

* En caso de una falla, tener alarmas para poder corregirla lo antes posible.

### **Nice to have**

## Optimización de costos

* Para que la empresa pueda ver si está gastando bien sus recursos.

## Trazabilidad de versiones end-to-end

* Necesitamos compatibilidad entre las versiones para que nuestra aplicación no tenga problemas al pasar de una versión a otra.

## Transparencia de ubicación de los servicios

* Al no saber a qué servidor te estás conectando, el equipo que maneja el servidor tiene la flexibilidad de cambiarte de uno hacia otro sin que te enteres.

## Proceso de deployment simple

* Para tener el menor costo de tiempo y dinero al deployar una versión.

# **Diseño de la Arquitectura**

## Amazon Route 53:

* Registro de Dominio
* Enrutamiento DNS: Enrutar el tráfico de Internet a la infraestructura que utiliza Amazon Web Services, como instancias Amazon EC2 o buckets S3.
* Health checking : Envía periódicamente solicitudes al endpoint para verificar que esté disponible, accesible y funcional.
* Razón fundamental para usar esto: Si su sitio se vuelve lo suficientemente grande como para tener varios servidores, querrá tener una automatización de sistemas, y poder automatizar los cambios de DNS facilita la misma.
* **Transparencia de ubicación de los servicios**

## Amazon CloudFront:

Está diseñado para administrar sus cargas de trabajo de video bajo demanda y en vivo. Una de nuestras condiciones es poder soportar una gran cantidad de datos multimedia. Cloudfront acelerará la entrega de los objetos estáticos o de los contenidos dinámicos a los clientes.

* Optimiza la latencia y baja la carga en los servidores principales.
* Posible reducción del costo de transferencia de datos. Dado que los datos no siempre se transfieren desde el mismo bucket de S3 en una región de AWS en particular.
* Más económico que otros CDNs (Como Akamai que cuesta $3500 por 10TB comparado a $1100 de CloudFront).
* Se integra con S3 Bucket y con EC2 fácilmente.
* Si tenemos inversores extranjeros, queremos que su experiencia de usuario sea la mejor posible.
* **Necesidad de streaming de información multimedia**

## Amazon S3:

El bucket S3 nos permite la escalabilidad que necesitamos sin ninguna complicación. En casos de algún error o desastre natural, gracias a la arquitectura de la misma, podemos restablecer el servicio gracias a su replicación entre regiones. También siendo uno de los líderes de la industria con esta plataforma podemos optimizar los costos de almacenamiento a largo plazo. Por principio usamos el **storage class de standard** en la parte de development (porque es free tier) y una vez adquirido una fuente de inversión, usaremos el storage class Intelligent, ya que está diseñado para optimizar los costos de almacenamiento moviendo automáticamente los datos al nivel de acceso al almacenamiento más rentable, sin impacto en el rendimiento ni en los gastos operativos.

* Backups
* Compatibilidad con RDS y CloudFront
* Escalable
* Los datos están disponibles cuando se necesitan y están protegidos contra fallas, errores y amenazas.
* **Soportar una cantidad grande de información multimedia**

## Amazon Elastic Beanstalk:

Es un servicio fácil de utilizar para implementar y escalar aplicaciones web. Soporta múltiples políticas de implementación, como la estrategia de blue/green deployments (zero downtime deployments), donde se crean dos entornos separados pero idénticos. Un entorno (azul) ejecuta la versión actual de la aplicación y un entorno (verde) ejecuta la nueva versión de la aplicación. Además, nos permite monitorizar y administrar el estado de sus aplicaciones.

* **Zero downtime deployments**

## Amazon Application Load Balancer:

* Distribuye el tráfico entrante de aplicaciones entre varios destinos como instancias de EC2
* Monitorea el estado de los destinos registrados, por lo tanto solo envía solicitudes a los destinos en buen estado.
* **Transparencia de ubicación de los servicios**

## PostgreSQL:

Para almacenar los datos necesitamos una base de datos relacional, ya que cuando hagamos las querys queremos poder hacer ‘joins’ entre las diferentes tablas. Esto se haría por ejemplo para filtrar los resultados de una búsqueda de propiedades con las características seleccionadas por el usuario. Dentro de las mejores opciones que hay, y las que más conocemos, se encuentran PostgreSQL y MySQL. Esta última la descartamos, ya que está destinada a proyectos más chicos, con menos datos.

La otra razón por la que elegimos PostgreSQL es que cuenta con una extensión llamada PostGIS que le agrega 3 características importantes a la base de datos. Estas son: tipos de datos espaciales, índices espaciales y funciones que operan sobre ellos. Esta base la vamos a usar para almacenar los datos geoespaciales, que son lo que utilizamos luego para el machine learning.

Otra opción muy reconocida en el mercado para almacenar datos geoespaciales es DynamoDB. La principal razón para no elegir esta última es que hacer querys complejas en una base de datos no relacional es mucho más difícil que en una sql.

* **Necesidad de guardar grandes cantidades de datos**

## Amazon Relational Database Service:

Necesitamos RDS para poder configurar, acceder y operar sobre la base de datos cloud. También lo vamos a usar para hacer los backups diarios al sistema, estos a su vez se guardan en el S3.

* **Necesidad de Backups (remotos)**

## Aws Lambda:

Permite ejecutar código sin aprovisionar ni administrar servidores. La vamos a usar para descargar los datos geoespaciales, por lo que es perfecto para este caso, ya que se ejecuta cada un mes. De esta forma no gastamos procesamiento en nuestros servidores, y nos resulta más barato.

* **Necesidad de poder conectar con distintos servidores.**

# **Machine Learning:**

## Python:

Dado que planeamos hacer un algoritmo de xgboost para el desarrollo del machine learning, seleccionamos como las mejores librerías a Pandas, Scikit-learn y Numpy. Todas estas librerías funcionan juntas con distintos propósitos. Numpy trabaja con números y permite al usuario hacer importantes cálculos con los datos sin necesidad de mucho código. Por otro lado, Scikit-learn es la librería más elegida para Machine Learning, ya que cuenta con algoritmos de ML que se pueden utilizar con pocos ajustes. Otra librería muy conocida es Pylearn2, pero no elegimos esta porque se recomienda para usuarios casi expertos en ML, mientras que Scikit-learn actúa como un ‘black box’ en la que el usuario no tiene que entender 100% lo que está pasando.

* **Poder operar sobre grandes cantidades de datos**
* **Soportar grandes procesamientos de datos**

## Amazon Elastic Container Registry:

El Amazon ECR sirve para poder controlar fácilmente las imágenes de los contenedores creadas en Docker en la plataforma de AWS. Esto también hace que se **paguen solamente los recursos que se usan.** A su vez ayuda en la seguridad de los deployments asegurándose que solamente tenga la posibilidad de hacer pull de imágenes con autenticacion y autorizacion. Además, está la posibilidad de hacer escaneos de imágenes para verificar si hay vulnerabilidades de seguridad conocidas.

(Amazon ECR elimina la necesidad de operar sus propios repositorios de contenedores o preocuparse por escalar la infraestructura.)

* Facilita nuestro deployment en instancias.
* Optimiza costos
* **Proceso de deployment simple**

## Docker:

Vamos a utilizar las imágenes de Docker para poder deployar a las instancias a través del ECR, de manera óptima y segura. Consta con mucha documentación y una comunidad grande.

- Documentación Amplia

- Haciendo containers facilitamos el deployment de nuestros servicios.

* **Proceso de deployment simple**

# **Monitoreo:**

## Amazon CloudWatch:

Servicio de monitoreo para mejorar el rendimiento operativo y optimizar los recursos. Nos permite definir alarmas y realizar acciones automatizadas como por ejemplo responder a cambios de rendimientos para mantener en ejecución la aplicación sin que esta se sobrecargue. Aprovechamos además la fácil interoperabilidad.

Vamos a utilizar los eventos de CloudWatch para iniciar las lambdas periódicamente para poder así actualizar los datos geoespaciales.

* **Observabilidad**
  + **Logging**
  + **Monitoreo**
  + **Alarmas**
* **Optimización de costos**

# **FrontEnd:**

## Spring:

Framework web que funciona sobre la JVM, uno de los entornos más extendido y potente. Tiene documentación bien estructurada y completa, además de un gran ecosistema, por lo que es rápido desarrollar y también rápido de ejecutar. Está optimizado para altas cargas de trabajo con un mínimo consumo de memoria, gracias a que la mayoría de entidades son singletons y por tanto objetos java reutilizados entre threads.

Otra razón importante por la que usamos Spring es por la seguridad. La otra opción para manejar este tema sería usar la Lambda de Amazon, pero esta tiene el problema de que eventualmente nos saldría más cara que levantar los containers de Docker, ya que se paga por requests en vez de tiempo corriendo.

* **Necesidad de tener una aplicación Web**

## Kotlin:

Tiene una sintaxis mucho más compacta comparada con Java y su sistema de tipos permite un código sin referencias nulas, por lo que brinda menos bugs, y por lo tanto un menor tiempo de desarrollo. Además es 100% interoperable con Java y comparable en desempeño con Java, por lo que va muy bien con Spring.

* **El soporte/mantenimiento debe ser lo más simple posible.**

# **BackEnd:**

## React:

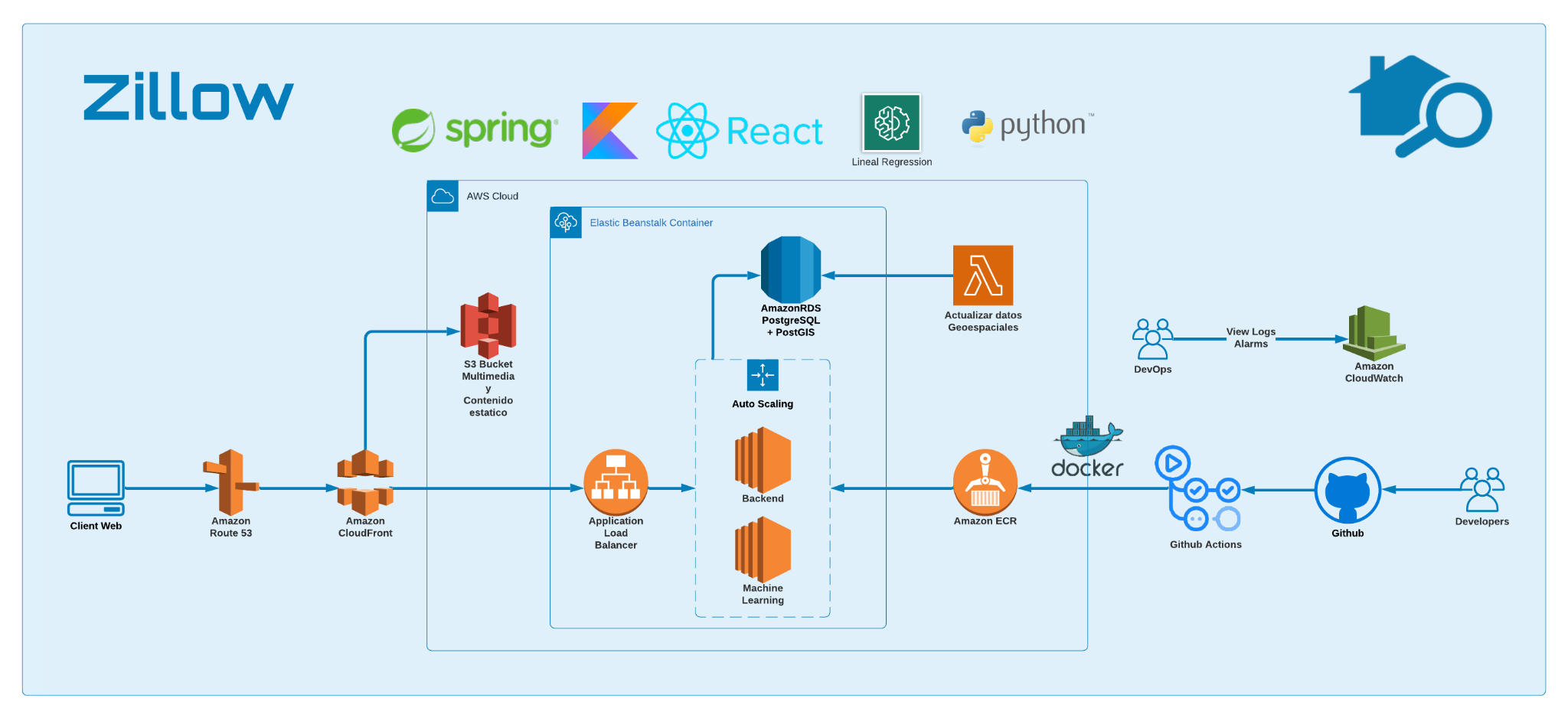
Esta librería es usada por ZonaProp y Zillow, que son aplicaciones similares a la nuestra, por lo que debería servir perfectamente para nuestros requerimientos. Usa un DOM virtual, cambiando solo lo necesario, por lo que brinda una mejor experiencia de usuario, además de un rendimiento y una fluidez impresionante. Además, aporta una facilidad para organizar el código, lo que a su vez brinda un desarrollo más rápido y escalable para crear nuevas funcionalidades. Por último, es fácilmente extensible para hacer una Progressive Web App, lo cual la hace más eficiente para usar en dispositivos móviles de gama baja, ya que cachea el cliente.

* **Sitio liviano para dar respuesta a celulares de gama baja**

## TypeScript:

Brinda un desarrollo más rápido, ya que evita muchos errores en compilación. Hace que el desarrollo sea más comprensible, sin perder la agilidad y las ventajas que tiene javascript.

* **El soporte/mantenimiento debe ser lo más simple posible.**



# **Estructura de Costos**

## Amazon CloudFront:

Free tier:Tenemos 12 Meses gratis con 50 GB de data de salida y 2.000.000 de HTTP/HTTPS Requests por mes.

Una vez pasado esto, se calcula 0.110US$ por GB de transferencia de datos de salida y 0.125US$ por GB de transferencia de datos de entrada.

Además 0.0160US$ por 10.000 HTTP requests (16 \* 10.000.000) y 0.0220US$ por 10.000 HTTPS requests. (220 \* 100.000.000)

Inicio de página: 2MB -> 100 requests

Búsqueda: 1MB por 6 props -> 100 requests

1 Propiedad = 5MB -> 150 requests

1 Inicio de página + 3 Busquedas + 12 Entradas de Propriedades

2MB + 3\*1MB + 12 \* 5MB = 65MB Por usuario -> 0,007 US$

En 200.000 usuarios -> 1430 US$

Requests:

100 + 3\*100 + 150\*12 = 2200 requests por usuario

En 200.000 \* 2200 = 440.000.000 requests HTTPS -> 968 US$

Costo mensual -> 1430 US$ + 968 US$ = 2398 US$ mensuales

## Amazon CloudWatch:

Vamos a comenzar a usar Amazon CloudWatch de forma gratuita. La mayoría de las métricas de ventas de los servicios de AWS están disponibles automáticamente de manera gratuita en CloudWatch.

Lo utilizamos para obtener métricas de monitoreo básico cada 5 minutos, para las alarmas y los logs de datos (hasta 5GB).

No existe ningún compromiso inicial ni tarifa mínima. Solo paga por lo que use y, a final de mes, se le cobrará el consumo.

Amazon brinda algunas estimaciones:

Cargos mensuales de CloudWatch para monitoreo detallado de EC2 (2 instancias) = 5 USD al mes

Cargos mensuales de CloudWatch para alarmas con detección de anomalías = 1,50 USD al mes

Cargos mensuales de CloudWatch para métricas de solicitudes a S3 = 0,10 USD por mes

## Amazon Elastic Beanstalk:

No se aplican cargos adicionales por utilizar AWS Elastic Beanstalk.

## Auto Scaling Group:

No se aplican cargos adicionales por utilizar Auto Scaling Group.

## Amazon Route 53:

12 US$ el dominio anual

50 US$ Traffic Flow

0,50US$ por zona alojada al mes para las primeras 25 zonas alojadas

0,40US$ por millón de consultas, el primer millardo de consultas mensual

0,20US$ por millón de consultas, superado el primer millardo de consultas mensual

Si tenemos 200.000 consultas mensuales, pagamos solo 0.4US$ por mes. Además nos sale 0.5US$ por tener la zona alojada y 1US$ mensual por dominio.

Aproximadamente planeamos gastar 55 US$ mensuales.

## Amazon Application Load Balancer:

Con el free tier se obtienen 750 horas por mes gratis.

Luego, cuando se termine el free tier, se estima un costo de 16US$ por mes para mantenerlo. Ya que se paga 0.0225US$ por hora y estimamos tenerlo corriendo aproximadamente 750 horas al mes.

## Amazon Elastic Container Registry:

Free tier: 500MB de almacenamiento durante 12 meses.

Calculamos necesitar 1GB por mes de IN y 1GB por mes de OUT con 10GB de storage. Esto se aproxima en 1 US$ mensual.

## Amazon RDS:

Free tier: 750 horas para usar una instancia de Single-AZ db.t2.micro corriendo PostgreSQL.

20GB de almacenamiento de propósito general (SSD)

20GB de almacenamiento para los backups automáticos y snapshots

Para guardar los datos de las propiedades y los geoespaciales en PostgreSQL necesitamos 30GB de storage por mes con 30GB adicionales de backup. Con una instancia de db.t3.micro termina costando unos 70 U$D.

## EC2:

Free tier:750 horas de instancias de Linux o Windows t2.micro, por cada mes durante un año.

Estimamos 100GB de almacenamiento. También calculamos que vamos a tener aproximadamente 1 TB por mes de transferencia de datos a internet y 1TB por mes de transferencia de datos entre regiones. Vamos a hacer snapshots al EBS 2 veces al día y va a almacenar 100GB de datos.

El workload va a ser de lunes a viernes y necesitamos que al menos haya 2 instancias 8:30 horas al día.

El total de todo esto sería 272 US$ al mes.

## EBS:

Free Tier:30GB Storage, 2 Millones I/Os, 1Gb Snapshot.

100GB de General Purpose = 0.152$\*100 = 15.2 US$ p/mes

600GB de Snapshots (6 Versiones) = 40.8 US$ p/mes

Aproximamos gastar 56 US$ mensualmente.

## S3:

Free Tier: 5 GB de almacenamiento de Amazon S3 Standard; 20.000 solicitudes GET; 2.000 solicitudes PUT, COPY, POST o LIST; y 15 GB de transferencia de datos cada mes durante un año.

Ya que debemos guardar las fotos de las publicaciones estimamos una necesidad de almacenamiento de 1TB mensual lo que nos costaría alrededor de 40 US$

## AWS Lambda

Free Tier: Tenemos 1.000.000 de requests gratis por la duración de este.

Después serán aproximadamente 0.2US$ cada 1M de requests y 0.0000166667US$ cada GB por segundo. Por lo que podemos decir que teniendo en cuenta que solo va a correr una vez por mes y estimamos que no va a correr más de 10 minutos, llegamos a un total estimado menor a 0.2US$ mensual.

## Total

Basándonos en las publicaciones creadas en páginas como zonaprop y argenprop, podemos estimar un flujo de creación de publicaciones de aproximadamente 40.000 inmuebles por mes. En lo que respecta a visitas a la página, estimamos unas 200.000 mensuales. Con estos datos podemos estimar un costo de 2900 US$ por mes.