Luis Maldonado

Prueba Data Engineer

Objetivo General: Diseñar e implementar una solución para procesar y analizar un conjunto de datos.

Estructura del proyecto:

Parte 1: Diseño de la solución

Frente a un proyecto real un error común es partir de un dataset específico disponible, para solucionar un problema vagamente definido. Lo ideal sería tener un problema claramente definido, y definir un proyecto alrededor de la necesidad y factibilidad de desarrollar un negocio a partir de la solución de dicho problema, y una vez se ha desarrollado mucho más la necesidad de negocio, la descripción del producto y el ajuste al mercado, ahí si se buscan los datos. Esto porque los datos son muy sensibles a cualquier sesgo, y en muchos casos los datos disponibles acumulan sesgos producidos por condiciones anteriores al proyecto y el proyecto va a propagar dichos sesgos hacia el futuro.

Entonces, para enfrentar el diseño de una solución, es fundamental tener un conocimiento claro de los requisitos de las características del sistema a implementar, para no correr el riesgo de diseñar para un requisito errado o bajo suposiciones de negocio equivocadas. Estos requisitos deben reducir la incertidumbre al máximo en los siguientes dominios: **Tiempo, Costo, Alcance, Calidad, Beneficios, Riesgos y Recursos**. Cualquier variación en estos dominios puede impactar enormemente las decisiones de diseño y la factibilidad del proyecto. Es por esto que usualmente se implementan controles: cronogramas, presupuestos, planes, procesos de control de cambio y de medición, SMART goals, KPIs, etc. Todas estas actividades son fundamentales en la planeación, tanto en proyectos de tipo predictivo (Waterfall) como en proyectos con enfoque iterativo (Agile).

Como estos requisitos no están disponibles y existe una limitación en tiempo y recursos, entonces este diseño básico se centrará en el desarrollo de un prototipo, asumiendo suposiciones para dichos requerimientos del sistema.

Definición del problema: Implementar dashboard con estadísticas descriptivas de los negocios de un área geográfica específica, soportado por un pipeline de datos, una base de datos y una arquitectura escalable en la nube.

Selección de fuente de datos:

La fuente de datos elegida es la base de datos pública de Negocios registrados en la ciuda de San Francisco, que esá publicada en la siguiente URL: https://data.sfgov.org/Economy-and-Community/Registered-Business-Locations-San-Francisco/g8m3-pdis/about_data

Esta base de datos se acccede a trvés de una API proveída por Socrata en el endpoint: https://data.sfgov.org/resource/g8m3-pdis.json

Se seleccionó esta base de datos porque contiene información similar a la que se espera para la posición a la que estoy aplicando en cuanto a que se tienen las ubicaciones geográficas y clasificatoria de 300K negocios de un área geográfica específica. Los datos son suficientemente diversos y representativos del problema a enfrentar. También son bastante heterogéneos en cuanto a los datos disponibles o ausentes para diferentes negocios, como a las características estadísticas de cada atributo presente en el dataset, lo cual es esperable en cualquier escenario real.

Diseño de esquema de base de datos:

Despues de un análisis de la base de datos, el cual se puede consultar en el documento *Data_Analysis.ipynb*, se encontraron estas características del conjunto de datos:

Datos estructurados: el conjunto de datos parece estar estructurado, con columnas y relaciones bien definidas. Es probable que cada fila represente información sobre una empresa sin datos repetidos ni ambiguedad.

Múltiples fuentes y rangos de tiempo: el conjunto de datos se ha recopilado de diferentes fuentes durante un siglo y abarca un largo período. Esto puede provocar variaciones en la calidad de los datos y la presencia de valores faltantes.

Teniendo en cuenta el conjunto de datos, podría diseñar una base de datos relacional con tablas para entidades clave como "Negocio", "Ubicación" y "Vecindario", teniendo en cuenta que los negocios pueden haber cambiado de ubicación en el tiempo y que también la definición y características de los vecindarios puede variar. Se utilizan claves foráneas para establecer relaciones. por ejemplo:

Tabla de Negocios:
- BusinessID (Primary Key)
- BusinessAccountNumber
- OwnershipName
- … (Todos los demás atributos del negocio)

Tabla de ubicaciónes:
- LocationID (Primary Key)
- BusinessID (Foreign Key)
- StreetAddress
- City
- State
- …(Todos los demás atributos de la ubicación)

Tabla de Vecindarios:
- NeighborhoodID (Primary Key)
- NeighborhoodName
- …

Para el caso propuesto específicamente, los datos a almacenar podrían estar relacionados con las campañas de mercadeo que hace cada negocio, por lo tanto una **base de datos no estructurada noSQL como MongoDB, tiene más sentido** ya que permiten una mayor escalabilidad horizontal y están diseñadas para manejar datos no estructurados o semiestructurados, con diferente integridad de datos y esquemas dinámicos o flexibles. MongoDB, en particular, permite estructuras de datos dinámicas y anidadas. Esta flexibilidad es beneficiosa cuando se trata de conjuntos de datos en los que pueden faltar campos o tener campos diferentes. Algunas bases admiten fragmentación para lograr

escalabilidad horizontal, lo cual podría ser beneficioso si es necesario hospedarse en un cluster de equipos.

Diseño de esquema de base de datos noSQL:

```
{
  "_id": ObjectId,
  "location_id": Number,
  "business_account_number": Number,
  "ownership_name": String,
  "dba_name": String,
  "address": {
    "street": String,
    "city": String,
    "state": String,
    "source_zipcode": String,
    "mail_address": String,
    "mail_city": String,
    "mail_zipcode": String,
    "mail_state": String
  },
  "dates": {
    "business_start_date": Date,
    "business_end_date": Date,
    "location_start_date": Date,
    "location_end_date": Date
  },
  "naics": {
    "code": String,
    "description": String
  },
  "taxes": {
    "parking_tax": Boolean,
    "transient_occupancy_tax": Boolean
  },
  "districts": {
    "supervisor_district": Number,
    "neighborhoods_analysis_boundaries": String,
    "business_location": String,
    "unique_id": String,
    "sf_find_neighborhoods": Number,
    "current_police_districts": Number,
    "current_supervisor_districts": Number,
    "analysis_neighborhoods": Number,
    "neighborhoods": String
  }
}
```

Esta base de datos podría ser hospedada de forma dockerizada, en caso que se requiera portabilidad, encapsulamiento, por ejemplo para uso en dispositivos móviles, y escalabilidad en términos de correr múltiples instancias, por ejemplo una instancia para cada negocio.

Sin embargo, para este caso de uso yo consideraría que la persistencia de los datos, y la necesidad de tener una base central para alimentar procesos de inteligencia de negocios y entrenamiento de ML, así como para proveer servicios de IA a cada negocio basados en la colección de datos centralizada, yo preferiría un entorno basado en la nube, el cual podría tanto en lotes como en tiempo real mediante ***funciones Lambda.*** El uso de funciones lambda permite la mayor flexibilidad para escalar automáticamente la capacidad del

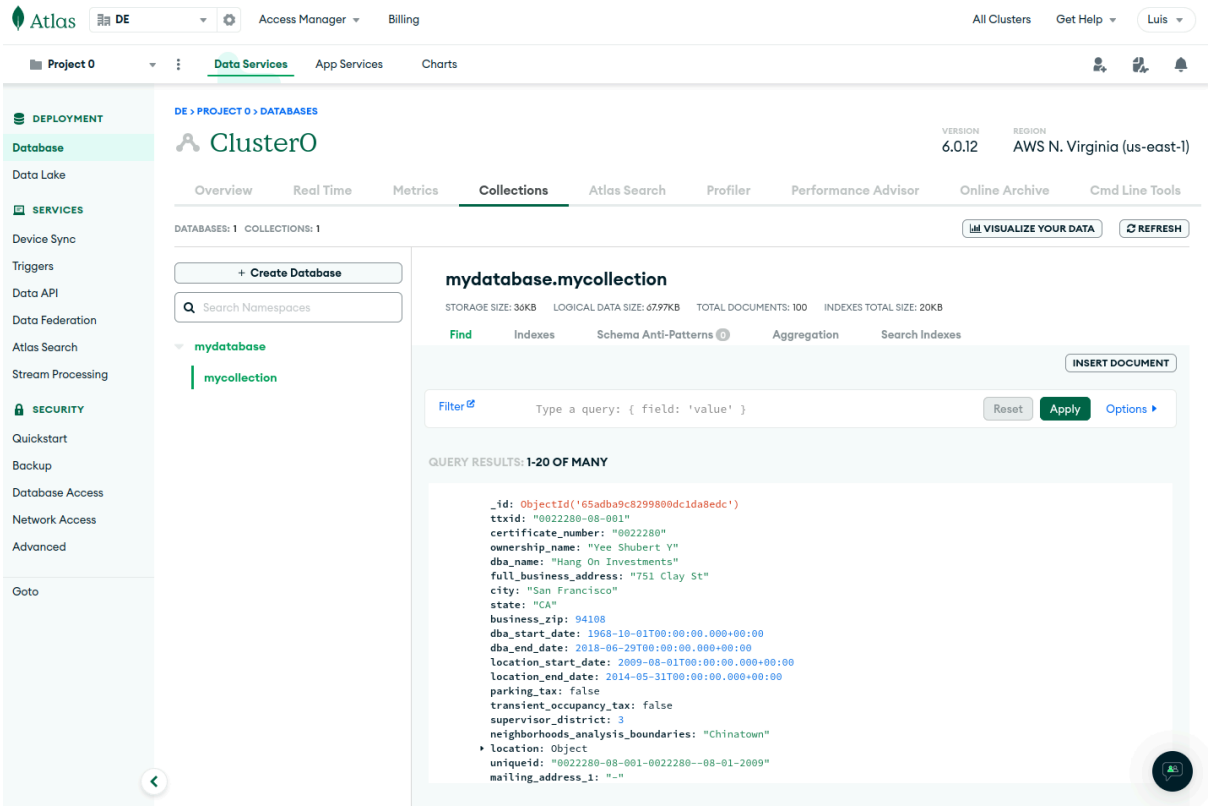
sistema, y pagar apropiadamente por la escala y el uso. Un entorno basado en la nube permitiría que diferentes consumidores puedan acceder a datos consistentes y actualizados.

Se usaría procesamiento por lotes en los casos en que los datos son cargados en volúmenes grandes y se espera que el tiempo de procesamiento sea grande también, por ejemplo en entrenamiento de algoritmos de ML. Por ejemplo para cargar los datos en el almacén de datos, hacer procesos de limpieza, transformación y analítica.

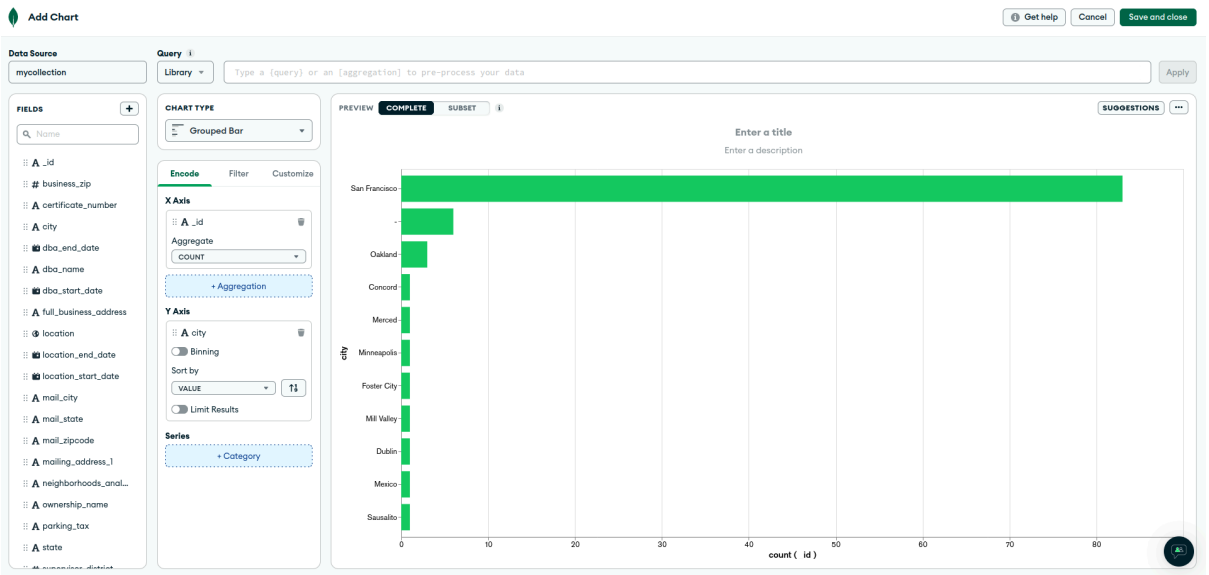
El procesamiento en tiempo real se requeriría para procesar peticiones y obtener respuestas con mínima latencia, basadas en datos frescos, por ejemplo disparadas por el usuario en una aplicación, por eventos o por cambios en el flujo de datos.

Implementación del almacén de datos:

para esta etapa es necesario definir las necesidades en términos de velocidad de procesamiento de datos, la escalabilidad y la confiabilidad esperada, cuales son las fuentes de datos, los destinos y las transformaciones necesarias. Como éste es un prototipo, Lo que se hizo fue montar los datos en un cluster MongoDB gratuito el cual puede adaptarse en el futuro para escalar el sistema o puede usarse en modo serverless también.



Mongo DB helps in previsualizing some statistics about the data:



Parte 2: Procesamiento y análisis de datos

Canalización ETL:

El módulo ***pipeline.py*** implementa los métodos para capturar los datos desde la API inicial, procesar y limpiar los datos haciendo unas cuantas operaciones sencillas y subir los datos a MongoDB. Este modulo implementa los métodos de los módulos **data_fetcher.py, data_cleaner.py** y **mongodb_updater.py**, los cuales son muy sencillos, pero pueden ser fácilmente complementados y mejorados.

Análisis de datos exploratorios (EDA):

En el Notebook ***Data Analysis.ipynb*** se muestra el ejemplo de un proceso muy sencillo de análisis de datos, generación de gráficas y algunas medidas estadísticas.

Análisis mas extenso:
Dependiendo de los objetivos de análisis, a este proceso se le pueden adicionar realizar pruebas estadísticas, crear modelos predictivos o crear visualizaciones más avanzadas. Este análisis más avanzado se dejó por fuera de este prototipo, ya que lleva bastante tiempo y la especificación de requisitos no lo incluye.

Parte 3: Optimización y escalabilidad

Cuellos de botella:

En cuanto a procesamiento, El pipeline es bastante rápido: Actualmente este pipeline sencillo, con 100 registros se demora aproximadamente unos 3 segundos en correr. Streamlit permite tareas sencillas, pero no es muy robusto para realizar grandes visualizaciones o manejar conjuntos pesados de datos.

Alternativamente se puede conectar un dashboard en PowerBI o Tableau, si se tienen las licencias.

En el tiempo de desarrollo de esta solución el cuello de botella más grande es el aseguramiento de que el formato de los datos que se capturan es consistente y que una vez almacenados en MongoDB y consultados por Streamlit, dichos datos no han cambiado de formato y siguen siendo los mismos.

La solución contempla el uso de mase de datos MongoDB en la nube y el uso de funciones Lambda para consultarla y servir los análisis a los usuarios. Sin embargo, frente a alternativas como una base dockerizada, sistemas SQL, o containerización y orquestación de servidores mediante docker y Kubernetes, presenta dos serias Limitaciones:

- Los procesos más pesados no deben consumir mucho tiempo ni recursos, para ser usadas por funciones lambda.
- La base de datos noSQL no es tan funcional si se requiere mantener las relaciones entre distintos tipos de datos de diferentes entidades.

Escalabilidad

Si se requiere capturar 100 veces más datos, lo primero que se debe considerar es por qué: ya que en la mayoría de casos, si se tiene un buen análisis o un buen muestreo, más datos no producen significativamente mejores resultados, y en cambio sí pueden incrementar la complejidad y la latencia de los procesos.

En este caso específico, teniendo datos de campañas de mercadeo de cada negocio, sí se presenta un gran flujo de datos, de gran diversidad, y se tiene un número muy grande de entidades nuevas, entonces sí se espera que el sistema tenga que escalar horizontal y verticalmente.

MongoDB permite el manejo de bases grandes y esquemas muy flexibles. A pesar de que puede ser un costo alto para 100.000 o 1M de registros. Se puede pagar el servicio serverless que cuesta USD $0.10 por 1 millón de lecturas, y hasta que se llegue a una base de datos de más de 100GB, considerar la migración al servicio dedicado. Este sistema es especialmente útil para proveer analítica a los numerosos e infrecuentes clientes de Orbiti.

El sistema es funcional para trabajar por lotes o en tiempo real con una carga muy baja y un dataset muy sencillo. Habría que hacer pruebas para ver su usabilidad en tiempo real, ver cuánta latencia se consigue.

Para una adecuada automatización, el script del pipeline de datos debe integrarse en una herramienta como Airflow, y una herramienta de CI/CD como Jenkins o GitLab.

Alternativas para el manejo de estos datos en lotes grandes podrían ser AWS Batch, con almacenamiento en AWS S3 o Google Cloud. Si la cantidad de datos crece bastante, consideraría que un entorno de desarrollo on-premises es más barato.

Una alternativa para el manejo de datos en tiempo real podría ser AWS Kinesis. Si el procesamiento es muy grande, sobre muchos datos, ya pensaría en implementar Apache Spark, aunque en ese caso, también esperaría un mayor tiempo de desarrollo de la solución, dada la complejidad del manejo de esa herramienta y todos los servicios, hardware y software que la soportan.

otras preocupaciones en la escalabilidad son:

- Balanceo de carga:

No es necesario usar un equilibrador de carga para distribuir las solicitudes entrantes entre varias instancias de su aplicación, ya que si se lanza la aplicación con funciones lambda cada una mantiene un uso mínimo de recursos y se paga por lo que consuma. El inconveniente es que cada instancia es relativamente pequeña y efímera.

- Arquitectura de microservicios:

se pueden implementar microservicios de forma independiente, lo que facilita el escalado de componentes individualmente, cada uno de los cuales maneja una funcionalidad específica.

- Fragmentación de bases de datos:

Para MongoDB, si la base de datos es demasiado grande, se puede considerar fragmentarla en varios servidores. La fragmentación puede distribuir los datos según una clave de fragmentación, lo que reduce la carga en un solo servidor. De nuevo, si se usa una arquitectura serverless en MongoDB, no se requiere fragmentación, pero se limita el tamaño de las bases de datos bastante.

- Optimizar la indexación de MongoDB:

Dependiendo de los nuevos datos, y cómo se usen en el sistema, se puede simplificar y acelerar el rendimiento de la base de datos si se optimizan los índices de la base, creando índices basados en los tipos de consultas más frecuentes o valiosos en la aplicación. MongoDb permite supervisar continuamente las consultas y los índices mediante el MongoDB profiler para identificar queries lentas.

- Pooling de conexiones:

Se puede implementar la agrupación de conexiones para administrar de manera eficiente las conexiones de bases de datos, reutilizándolas

- Mecanismos de tolerancia a fallos:

Implementar redundancia en varios niveles, incluidos servidores, bases de datos y componentes de todo el pipeline de datos.

Configurar replicación de la base de datos para MongoDB para crear múltiples copias de los datos y garantizar la disponibilidad de los datos incluso si una réplica deja de funcionar.

Implementar mecanismos de reintento para operaciones fallidas, como llamadas a API o escrituras en bases de datos. Considerar que el reintento puede abrumar los servicios durante una falla.

Implementar breakers, para bloquear temporalmente las peticiones a un servicio durante un fallo, o dependiendo de un evento en el flujo de datos.

Copias de seguridad periódicas y versionadas en ubicaciones, tecnologías, y sistemas diferentes y aislados. estos también podrían funcionar como equipos en reserva activa o pasiva.

Se supone que los clientes van a estar en una misma área geográfica de servicios en la nube, pero tal vez es útil tener servicios de backup funcionando en otra area de servicios (Norteamérica, por ejemplo)

Cifrar datos confidenciales que estén en tránsito o almacenados. Pero una mejor estrategia es no capturar datos confidenciales. También destruir datos obsoletos, para reducir la exposición y responsabilidad ante un fallo de seguridad.

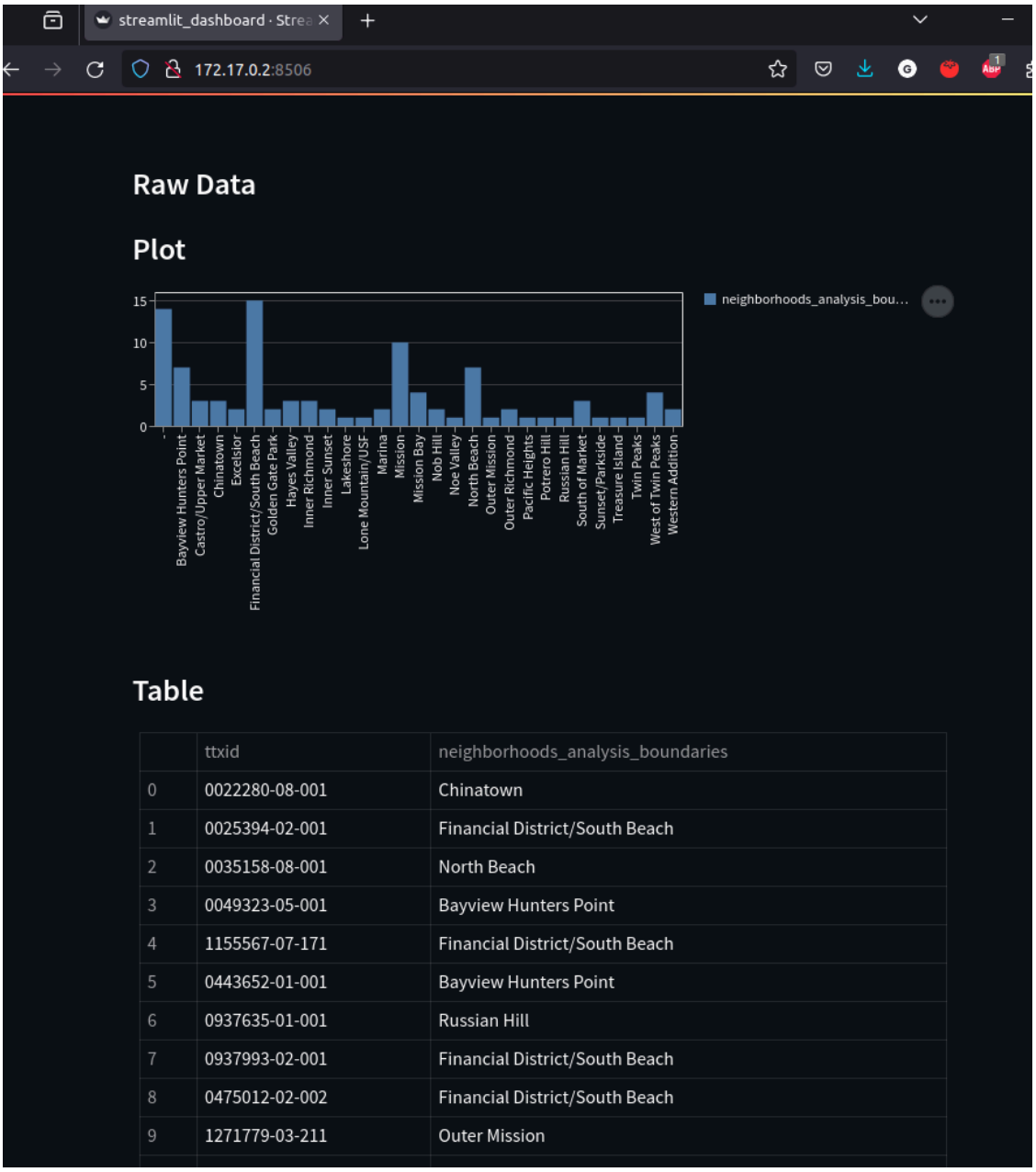
Tener un plan de respuesta a incidentes que se active automáticamente y que guíe a los miembros del equipo en caso de una emergencia. Considerar que la emergencia puede dejar fuera de capacidad al equipo mismo o a los sistemas automáticos.

Pruebas continuas, Simulacros, Ejercicios, o Caos Engineering pueden ayudar a probar la resistencia del sistema e identificar problemas.

Parte 4: Tableros interactivos:

Una herramienta rápida para crear tableros interactivos es Streamlit. La ventaja que tiene es que es muy liviana y portable. Así como el pipeline de datos es escalable, el tablero se puede lanzar en cualquier navegador web, o embebido en otra página web y se pueden crear múltiples instancias del mismo. Del mismo modo, se puede programar de forma que el tablero a mostrar sea diferente dependiendo de algun evento en el pipeline.

Dado el tiempo para la prueba, el tablero es muy sencillo y muestra una sola gráfica con el número de negocios en cada vecindario de San Francisco, a partir de una muestra de los datos, que se capturan de la base de datos en MongoDB. Actualmente el tablero se lanza desde el pipeline, se despliega por 30 segundos y luego se cierra el proceso automáticamente.



	ttxid	neighborhoods_analysis_boundaries
0	0022280-08-001	Chinatown
1	0025394-02-001	Financial District/South Beach
2	0035158-08-001	North Beach
3	0049323-05-001	Bayview Hunters Point
4	1155567-07-171	Financial District/South Beach
5	0443652-01-001	Bayview Hunters Point
6	0937635-01-001	Russian Hill
7	0937993-02-001	Financial District/South Beach
8	0475012-02-002	Financial District/South Beach
9	1271779-03-211	Outer Mission

Parte 5: Documentación e implementación

Documentación de la arquitectura del sistema

Herramientas y procesos elegidos:
El sistema es un pipeline muy simple en python que lee datos desde la API Socrata, los procesa en Pandas y luego los sube a una base en MongoDB. El proceso dura aproximadamente 1.7 segundos para capturar 10.000 datos y subir una muestra de 100. Adicionalmente el pipeline lanza un dashboard en streamlit por 30 segundos. El lanzamiento se realiza desde un contenedor docker y dura aproximadamente 2 segundos.

Entorno de implementación:
Se utilizó un entorno virtual de desarrollo en Python, de manera que se logra encapsular las librerías requeridas.
Se usó mi cuenta personal de MongoDB para crear un proyecto nuevo y asociado a éste un cluster compartido (gratuito). No se configuró seguridad por SSH, que sería deseable, solamente se bloqueó el acceso a IPs distintas a mi equipo.

El código fuente se encuentra en el siguiente repositorio público en GitHub:
https://github.com/l-maldonado/Project_SFBusiness

Instrucciones de implementación:

Se debe tener un entorno de implementación en python, una alternativa es usar un entorno containerizado en docker, el cual se puede crear mediante el siguiente código:

```
docker run -p 8888:8888 -v /media/alien/WinHD/DS4A:/home/jovyan/work
                                        \jupyter/scipy-notebook:17aba6048f44
docker exec -it 28fc92619d17 sh
```

esto da acceso a la consola sh del contenedor, en la cual se puede correr Jupyter notebooks:

```
>>jupyter notebook list
```

Y se puede cerrar el proceso:

```
jupyter notebook stop 8888
```

Se pueden instalar todas las librerías listadas en requirements.txt, pero son muchas, alternativamente, las librerías críticas para este proyecto se pueden instalar así:

```
pip install pandas matplotlib seaborn missingno folium geopy.geocoders pymongo streamlit
-m pip install "pymongo[srv]"
```

Además de ello, se necesita un servicio de base de datos corriendo. Específicamente se sua MongoDB atlas. Se debe crear un cluster, una base de datos y una colección. Los datos de acceso se deben ingresar en el código. Streamlit proporciona la gestión de datos secretos, para no exponer las contraseñas.

Test Unitarios

En la carpeta Tests, del código fuente, se encuentran lso test unitarios de cada módulo. Sin embargo, por cuestiones de tiempo, estos no se han terminado de realizar. Se pueden correr los test unitarios asi:

```
python -m unittest discover tests
```