Informe de Arquitectura de Datos para SolidarianID

Arquitectura de Datos

Máster Universitario en Ingeniería del Software

Autores:

Hernán Salambay Roldán
Alejandro Montoya Toro
Pedro Nicolás Gomariz
Aurora Hervás López
Dongyue Yu

15 de Enero de 2025

Facultad de Informática
UMU

Contenidos

1 Introducción.

Este documento describe la arquitectura de datos del sistema SolidarianID, una plataforma orientada a la participación y gestión de causas solidarias mediante comunidades colaborativas. La arquitectura se ha diseñado siguiendo un enfoque distribuido, basado en microservicios independientes que gestionan distintos subdominios: Usuarios, Comunidades y Estadísticas. La comunicación entre estos microservicios se realiza de manera asíncrona mediante un **Broker de Mensajería Kafka**, garantizando flexibilidad, escalabilidad y consistencia eventual.

El objetivo principal de esta arquitectura es proporcionar un sistema robusto y escalable que facilite el acceso y la manipulación eficiente de los datos relacionados con los usuarios, comunidades, causas solidarias y acciones específicas. Cada microservicio implementa su propia estructura de almacenamiento, utilizando los sistemas de bases de datos más adecuados para sus necesidades:

- **Microservicio de Usuarios:** Gestiona la información de los usuarios registrados, sus interacciones y su historial de acciones dentro de la plataforma, utilizando **PostgreSQL** con **TypeORM** para un manejo relacional y consistente.
- **Microservicio de Comunidades:** Administra las comunidades, causas y acciones solidarias, empleando **MongoDB** con **Mongoose** para un almacenamiento flexible basado en documentos.
- **Microservicio de Estadísticas:** Almacena información agregada y proporciona reportes sobre métricas clave del sistema, utilizando **Cassandra** para consultas rápidas y manejo eficiente de grandes volúmenes de datos distribuidos.

A lo largo del documento, se detalla el diseño conceptual de los datos, las justificaciones de las decisiones de almacenamiento, los esquemas implementados y los mecanismos de replicación, escalabilidad y tolerancia a fallos. También se describen las consultas más relevantes y las instrucciones para el despliegue de los microservicios, destacando el uso de contenedores Docker para una gestión eficiente de los entornos de ejecución.

2 Diseño e implementación de la arquitectura de datos.

La elección de los sistemas de almacenamiento en SolidarianID sigue un enfoque basado en los requisitos de cada microservicio, priorizando escalabilidad, consistencia y eficiencia de las operaciones según el tipo de datos y la carga esperada. En este apartado, se describen las bases de datos elegidas y la justificación de su uso.

2.1 Microservicio de Usuarios.

El Microservicio de Usuarios utiliza ***PostgreSQL*** como base de datos relacional junto con ***TypeORM*** debido a su capacidad para gestionar datos relacionados de manera consistente y eficiente. PostgreSQL permite definir claves foráneas y aplicar restricciones de integridad referencial, garantizando relaciones fiables entre entidades como perfiles de usuarios, historial de actividades, notificaciones y relaciones de seguidores/seguidos.

PostgreSQL soporta consultas avanzadas, como uniones y subconsultas complejas, manteniendo un buen rendimiento incluso en relaciones muchos a muchos. La integración con TypeORM añade una capa de abstracción orientada a objetos que simplifica la manipulación de datos y el mantenimiento del código.

Como alternativa, MongoDB ofrece alta disponibilidad y escalabilidad horizontal mediante su arquitectura distribuida, siendo ideal para aplicaciones que manejan grandes volúmenes de datos semi-estructurados. Sin embargo, al seguir un modelo de consistencia eventual, no garantiza la misma precisión en transacciones complejas, lo que puede ser una limitación en escenarios donde la integridad de los datos es crítica, como el registro e historial de acciones dentro de la plataforma.

Aunque PostgreSQL requiere mayor configuración para escalar horizontalmente, sigue siendo la mejor opción para operaciones críticas y transacciones consistentes. Por su parte, MongoDB es más adecuado para servicios enfocados en escalabilidad masiva y alta velocidad de lectura/escritura, siempre que no se requieran transacciones complejas.

2.2 Microservicio de Comunidades.

El Microservicio de Comunidades utiliza MongoDB como base de datos NoSQL debido a su flexibilidad y capacidad para manejar datos semi-estructurados y relaciones dinámicas entre entidades como comunidades, causas y acciones. Permite modelar datos complejos de forma eficiente mediante documentos JSON anidados, lo que resulta ideal para un dominio altamente relacionado y cambiante. Su esquema flexible facilita la evolución del modelo de datos sin afectar la continuidad del servicio, lo cual es esencial en aplicaciones donde las relaciones y estructuras pueden cambiar con frecuencia.

MongoDB es una solución equilibrada para este microservicio, ya que soporta tanto escrituras como lecturas con buen rendimiento, gracias a su arquitectura distribuida que permite escalabilidad horizontal. Además, sus índices avanzados y la capacidad para realizar agregaciones complejas permiten gestionar consultas eficientes, algo fundamental para manejar las relaciones entre comunidades, causas y acciones, y presentarlas de manera rápida en el frontend.

Otra alternativa barajada es Cassandra, que está optimizada para operaciones de escritura masiva y series temporales, pero no ofrece la misma flexibilidad para modelar relaciones complejas ni realizar consultas avanzadas. Redis, aunque es extremadamente rápido, no es adecuado para modelar datos relacionados debido a su naturaleza clave-valor y enfoque en memoria. Neo4j sobresale en manejar relaciones complejas, pero no tiene la flexibilidad ni el rendimiento generalizado en escrituras y lecturas que MongoDB ofrece para este caso.

Aunque MongoDB utiliza un modelo de consistencia eventual, lo que puede ser una limitación en escenarios con transacciones altamente críticas, sigue siendo la mejor opción en este microservicio debido a su equilibrio entre flexibilidad, escalabilidad y capacidad para manejar tanto escrituras como lecturas eficientes, asegurando un desempeño confiable y ágil para el manejo de comunidades, causas y acciones.

2.3 Microservicio de Estadísticas.

El microservicio de estadísticas utiliza Cassandra como base de datos NoSQL debido a su capacidad para gestionar grandes volúmenes de datos y su excelente rendimiento en operaciones intensivas de escritura. Este microservicio recibe datos cada vez que un usuario interactúa con el microservicio de comunidades, lo que requiere una base de datos capaz de procesar altas tasas de escrituras concurrentes. Gracias a su arquitectura distribuida maestro-maestro, Cassandra garantiza alta disponibilidad, escalabilidad horizontal y baja latencia en las escrituras.

Diseñada para manejar datos de series temporales, Cassandra es ideal para registrar eventos y métricas, priorizando el rendimiento sin comprometer la disponibilidad gracias a su modelo de consistencia ajustable. Su modelo distribuido organiza los datos en tablas estructuradas con claves de partición y clustering, lo que permite un acceso eficiente y optimiza las consultas necesarias para generar informes y gráficos en el frontend.

En comparación, MongoDB ofrece flexibilidad para manejar datos semi-estructurados, pero su consistencia eventual y limitaciones en escrituras masivas lo hacen menos adecuado para cargas intensivas. Redis, aunque extremadamente rápido, no es ideal para almacenar grandes volúmenes de datos históricos debido a su naturaleza en memoria y persistencia limitada. Por otro lado, Neo4j, diseñado para gestionar relaciones complejas, no maneja de manera eficiente flujos de escritura elevados ni datos de series temporales.

Aunque Cassandra presenta desventajas, como su consistencia eventual, la falta de soporte para consultas relacionales avanzadas, almacenamiento redundante y una curva de aprendizaje pronunciada, sigue siendo la mejor opción para este microservicio al garantizar acceso eficiente a los datos en tiempo real, optimizando la experiencia del frontend.

3 Diseño conceptual de datos.

En esta sección se detalla el diseño conceptual de datos del sistema SolidarianID, basado en un modelo distribuido compuesto por los microservicios de Usuarios, Comunidades y Estadísticas. Se presenta un diagrama conceptual de las entidades y relaciones más relevantes, seguido de una explicación de las entidades, sus atributos clave y las relaciones entre ellas.

3.1 Microservicio de Usuarios.

La entidad principal de este microservicio es ***User***, la cual representa a cada usuario mediante un identificador único (*id*), datos personales básicos (*firstName, lastName, email, birthDate*), y otros atributos opcionales como una breve descripción del perfil (*bio*). Para garantizar la privacidad, se incluyen flags (*showAge, showEmail*) que permiten al usuario decidir si su edad y correo electrónico son visibles públicamente. Además, se mantiene un conjunto de seguidores y seguidos, lo que crea una relación *muchos a muchos* entre los usuarios.

La entidad ***HistoryEntry*** almacena eventos que reflejan acciones relevantes que el usuario ha realizado en la plataforma, como solicitudes para unirse a comunidades o contribuciones a causas solidarias. Los atributos clave de esta entidad incluyen el identificador del usuario (*userId*) al que pertenece la entrada, el identificador de la entidad relacionada (*entityId*), como una comunidad, una causa o acción, el tipo de actividad (*type**), el estado de la actividad (*status***) y el *timestamp* para registrar la fecha y hora del evento. Esta entidad es crucial para mantener un registro detallado de las interacciones de los usuarios y puede ser consultada tanto por el propio usuario como por los administradores.

**Activity Type Values:*
COMMUNITY_ADMIN, JOIN_COMMUNITY_REQUEST_SENT, JOIN_COMMUNITY_REQUEST_REJECTED, JOINED_COMMUNITY, CAUSE_SUPPORTED, CAUSE_CREATED, ACTION_CONTRIBUTED, USER_FOLLOWED

***Entry Status Values: PENDING, APPROVED, REJECTED, ARCHIVED*

Por otro lado, la entidad ***Notification*** representa las notificaciones generadas cuando un usuario al que se sigue realiza una actividad relevante, como la creación de una comunidad o la contribución a una causa. Cuando se registra un evento en el historial, se recupera al usuario que ha generado dicho evento y se crea una notificación para cada uno de sus seguidores. Esta

notificación incluye información sobre el tipo de actividad (*activityType*), el identificador de la entidad asociada (*entityId*) y un indicador (*read*) para señalar si la notificación ha sido leída. Este enfoque permite al sistema gestionar eficientemente las notificaciones y mantener a los usuarios informados sobre las novedades relacionadas con las personas que siguen.

En este enfoque relacional, se utiliza *UUID* como identificador primario para cada entidad, lo que asegura unicidad y escalabilidad. Los tipos de datos clave incluyen *string* para los atributos de texto, *boolean* para indicadores de visibilidad y *date* para las fechas.

Las relaciones uno a muchos y muchos a muchos se implementan mediante tablas intermedias y claves foráneas, garantizando la integridad referencial y la consistencia en la gestión de los datos del sistema.

En la Figura 1, se muestra un diagrama entidad-relación (ER) que representa el modelo de datos correspondiente a este microservicio.

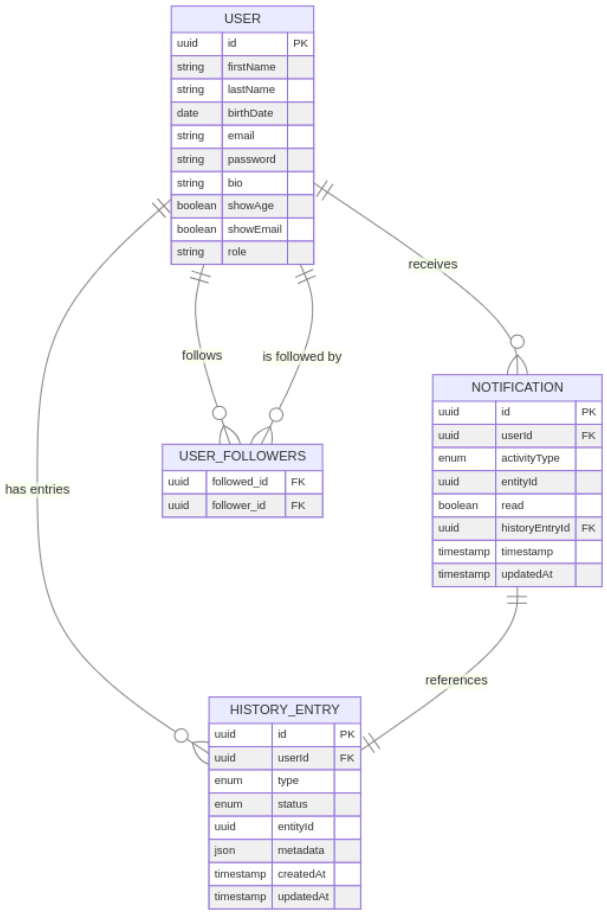

Figura 1. Diagrama Entidad-Relación Microservicio de Usuarios.

3.2 Microservicio de Comunidades.

El microservicio de Comunidades es responsable de gestionar las comunidades creadas en la plataforma, así como las causas solidarias y las acciones asociadas a ellas. La entidad principal es ***Community***, la cual representa un espacio en el que los usuarios pueden reunirse y colaborar en iniciativas de impacto social. Cada comunidad se identifica mediante un id único (*id*) y está compuesta por un nombre (*name*), una descripción (*description*) y un (*adminId*), que corresponde al usuario administrador encargado de gestionar la comunidad. Además, se

mantienen listas de usuarios que forman parte de la comunidad (*members*) y las causas activas promovidas por dicha comunidad (*causes*).

La entidad ***Cause*** está identificada por un (*id*) único y tiene atributos como título (*title*), descripción (*description*), fecha de finalización (*endDate*) y una lista de *ods*, que representa los Objetivos de Desarrollo Sostenible (ODS) a los que está vinculada la causa. Una causa pertenece a una única comunidad, indicada por el (*communityId*), pero puede tener múltiples (*actionsIds*), que corresponden a las acciones concretas que se realizan para cumplir los objetivos de la causa. Además, se mantiene un registro de (*supportersIds*), que representa a los usuarios que apoyan la causa.

La entidad ***Action*** representa actividades dentro de una comunidad, clasificadas en tres tipos: económica, recolección de bienes y voluntariado. Incluye atributos comunes como *type**, *status***, *title*, *description*, *target*, *achieved*, *causeId*, *communityId* y *createdBy*. Dependiendo del tipo de acción, se agregan campos opcionales: *goodType* para recolección de bienes y *location* y *date* para voluntariado.

**Cause Type Values: economic, goods_collection, volunteer*
***Status Values: PENDING, IN_PROGRESS, COMPLETED*

La entidad ***Contribution*** representa las contribuciones realizadas por los usuarios a las acciones solidarias. Cada contribución tiene un id único, junto con atributos clave como el identificador del usuario que realiza la contribución (*userId*), la acción a la que pertenece la contribución (*actionId*), la cantidad aportada (*amount*) y la unidad de medida, cómo euros, horas de voluntariado, etc. (*unit*). También incluye la fecha en que se realizó la contribución (*date*). Esta entidad permite registrar un seguimiento detallado de los esfuerzos colectivos en las acciones solidarias.

La entidad ***JoinCommunityRequest***, que representa peticiones enviadas por los usuarios para formar parte de una comunidad. Cada solicitud incluye atributos como el usuario que realiza la solicitud (*userId*), la comunidad a la que quiere unirse (*communityId*) y el administrador encargado de aprobar o rechazar la solicitud (*adminId*). Además, cuenta con un campo *status* para reflejar el estado de la solicitud (*PENDIENTE*, *APROBADA*, *RECHAZADA*) y un campo opcional *comment*, donde el administrador puede dejar observaciones, si rechaza dicha solicitud.

Por último, la entidad ***CreateCommunityRequest*** gestiona las solicitudes de creación de nuevas comunidades por parte de los usuarios. Esta entidad incluye un id único, el identificador del usuario que realiza la solicitud (*userId*), y atributos como nombre propuesto para la comunidad (*name*), descripción de la comunidad (*description*), y un campo *cause* que representa una causa inicial propuesta, con atributos como *title*, *description*, fecha de finalización (*end*) y lista de ODS relacionados (*ods*). La solicitud también incluye un *status* para indicar el estado de la petición (*PENDIENTE*, *APROBADA*, *RECHAZADA*) y un campo opcional *comment* para que el administrador deje observaciones sobre la solicitud en caso de rechazo.

En este enfoque basado en documentos, se utiliza *UUID* almacenado como *string* como identificador primario para cada entidad, asegurando unicidad y flexibilidad en la estructura de los datos. Los tipos de datos clave incluyen *string* para los atributos de texto, *boolean* para indicadores de estado, *array* para listas de IDs relacionadas y *date* para las fechas de creación y finalización.

Las relaciones uno a muchos y muchos a muchos se modelan mediante referencias explícitas dentro de los documentos, lo que permite consultas rápidas y eficientes sin comprometer la consistencia del modelo.

En la Figura 2, se presenta un diagrama entidad-relación (ER) que representa el modelo de datos correspondiente al microservicio de Comunidades, mostrando sus principales entidades y las relaciones entre ellas.

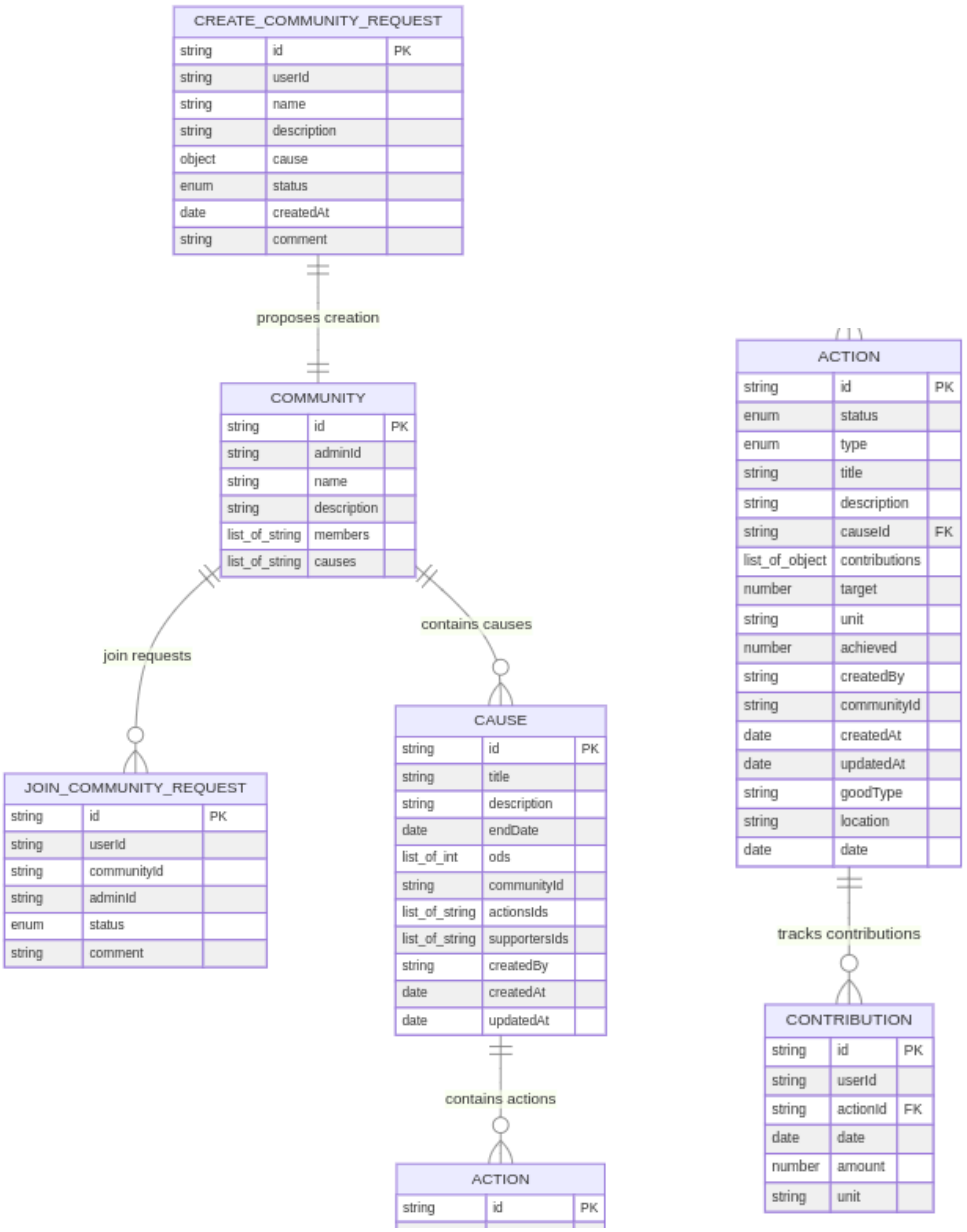

Figura 2. Diagrama Entidad-Relación Microservicio de Comunidades.

3.3 Microservicio de Estadísticas.

El microservicio de Estadísticas recopila y almacena datos agregados relacionados con las interacciones de los usuarios en la plataforma. Este modelo de datos está diseñado para resolver de manera óptima las consultas relacionadas con estadísticas y reportes de la plataforma, permitiendo obtener de forma eficiente información como el número de comunidades y causas asociadas a cada ODS, el porcentaje de apoyos por ODS y por comunidad, así como el progreso promedio de las acciones en cada comunidad. Además, facilita la generación de informes detallados sobre cada comunidad, incluyendo el número de miembros, la lista de ODS asociados a sus causas, el número de apoyos por causa, las acciones

realizadas y el progreso alcanzado en cada una de ellas, optimizando tanto la recuperación de datos individuales como los reportes agregados a nivel global.

La entidad ***OdsStatistics*** representa las estadísticas de la plataforma organizadas por Objetivos de Desarrollo Sostenible (ODS). Esta entidad incluye atributos como *odsId*, que identifica el ODS correspondiente, *communitiesCount*, que registra el número de comunidades vinculadas al ODS, *causesCount*, que refleja el número de causas asociadas, y *supportsCount*, que contabiliza el número de apoyos recibidos por las causas relacionadas. Esta tabla está diseñada para consultas rápidas que permitan visualizar el impacto de las causas y comunidades en relación con los ODS. Se utiliza solo la clave de partición *ods_id* ya que la tabla almacena una única fila por ODS. No se requieren columnas de clustering porque no se necesita un ordenamiento adicional dentro de una partición. La estructura de esta tabla permite consultas rápidas por *ods_id*, cómo "*obtener estadísticas generales de un ods específico*".

Para evitar contar una misma comunidad varias veces al calcular el número de causas asociadas a un ODS, se utiliza la entidad ***OdsCommunity***. Esta tabla actúa como una referencia que garantiza la unicidad de las comunidades relacionadas con un ODS, evitando duplicidades en las estadísticas. Sus atributos principales son *odsId*, que indica el ODS al que está vinculada la comunidad, y *communityId*, que identifica la comunidad correspondiente. La combinación de *ods_id* como clave de partición y *community_id* como clave de clustering permite realizar consultas eficientes como *"obtener todas las comunidades vinculadas a un ODS"*. Además, al definir *community_id* como clave de clustering, se asegura que no haya duplicados dentro de un ODS.

La entidad ***CommunityStatistics*** almacena información agregada sobre cada comunidad, con atributos como *communityId*, *communityName*, *supportCount* (número de apoyos recibidos en todas sus causas), *actionsTargetTotal* (suma de las metas definidas para todas las acciones) y *actionsAchievedTotal* (suma de las cantidades logradas). Esta entidad permite al administrador consultar datos clave de cada comunidad y evaluar el desempeño general de sus iniciativas. No se utilizan columnas de clustering porque cada comunidad se almacena de forma independiente, y no es necesario ordenar los datos dentro de una partición. La estructura de esta tabla permite consultas rápidas por *community_id*, como "*obtener estadísticas generales de una comunidad específica*".

La entidad ***communities_by_community_id*** almacena información detallada sobre cada comunidad registrada en la plataforma, diseñada para consultas directas basadas en el identificador único de la comunidad. Los atributos clave son *communityId*, *communityName*, *adminId* (administrador de la comunidad), *membersCount* (número de miembros de la comunidad), y *ods*, un conjunto de identificadores de ODS relacionados con las causas de la comunidad. Se utiliza *community_id* como clave de partición, lo que permite acceder rápidamente a la información de una comunidad específica mediante consultas como *"obtener la información básica de una comunidad por su ID"*. Al ser una consulta directa, no se requiere una clave de clustering.

La entidad ***causes_by_community*** agrupa todas las causas asociadas a una comunidad específica. Sus atributos principales son *communityId*, que identifica la comunidad a la que pertenece la causa, *causeId* (identificador único de la causa), *causeName* (nombre de la causa), *supportsCount* (número de apoyos recibidos por la causa), y *ods*, un conjunto de ODS vinculados a la causa. La clave de partición *community_id* permite agrupar todas las causas de una comunidad en una sola partición, mientras que la clave de clustering *cause_id* ordena las causas dentro de la partición. Esto optimiza las consultas para recuperar todas las causas de una comunidad en orden ascendente por su identificador. La estructura permite responder consultas como *"listar todas las causas de una comunidad específica"*.

La entidad ***actions_by_cause*** almacena información detallada sobre las acciones relacionadas con una causa. Sus atributos principales son identificador único de la causa (*causeId*),

identificador único de la acción (*actionId*), nombre de la acción (*actionName*), meta cuantificable de la acción (*target*) y cantidad alcanzada hasta el momento (*achieved*). La clave de partición *cause_id* agrupa todas las acciones de una causa en la misma partición, y la clave de clustering *action_id* ordena las acciones dentro de esa partición. Este diseño optimiza las consultas como *"obtener todas las acciones de una causa en orden ascendente por ID"*, facilitando la visualización del progreso de las acciones.

El modelo de datos del microservicio de Estadísticas utiliza un enfoque basado en columnas para manejar grandes volúmenes de datos con inserciones masivas y consultas optimizadas por claves de partición y se utiliza clustering para ordenar los resultados y acelerar las consultas. Esta estructura permite al administrador acceder rápidamente a métricas clave sobre el estado de las comunidades, las causas y las acciones solidarias de la plataforma.

En la Figura 3, se presenta un diagrama entidad-relación (ER) que representa el modelo de datos correspondiente al microservicio de Estadísticas, mostrando sus principales entidades y las relaciones entre ellas.

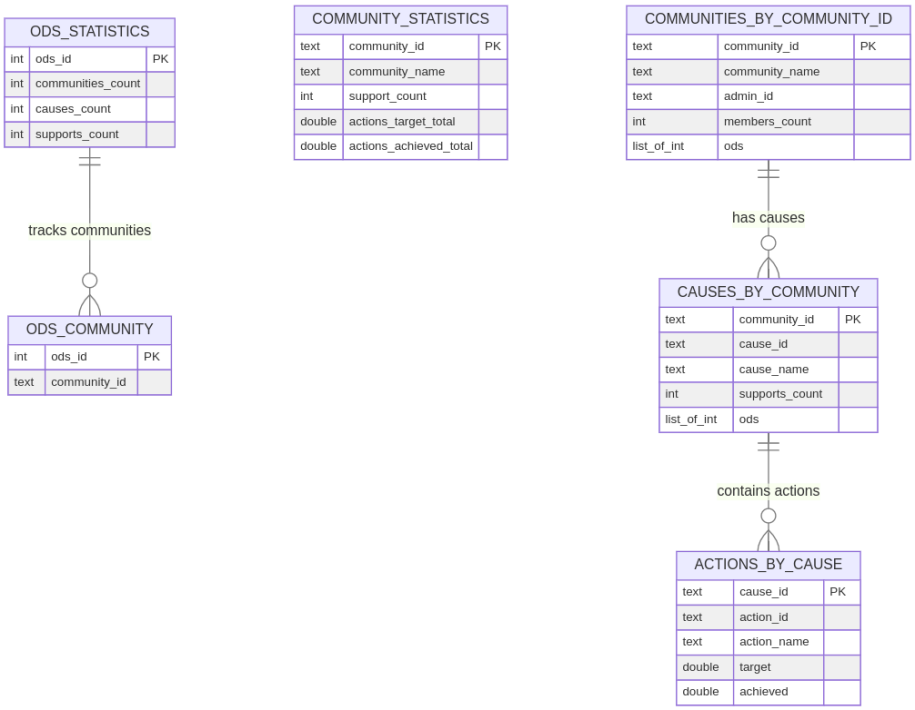

Figura 3. Diagrama Entidad-Relación Microservicio de Estadísticas.

4 Diseño de la arquitectura.

4.1 Diagrama de la arquitectura final.

El diagrama presentado en la Figura 4 describe la arquitectura distribuida de SolidarianID.

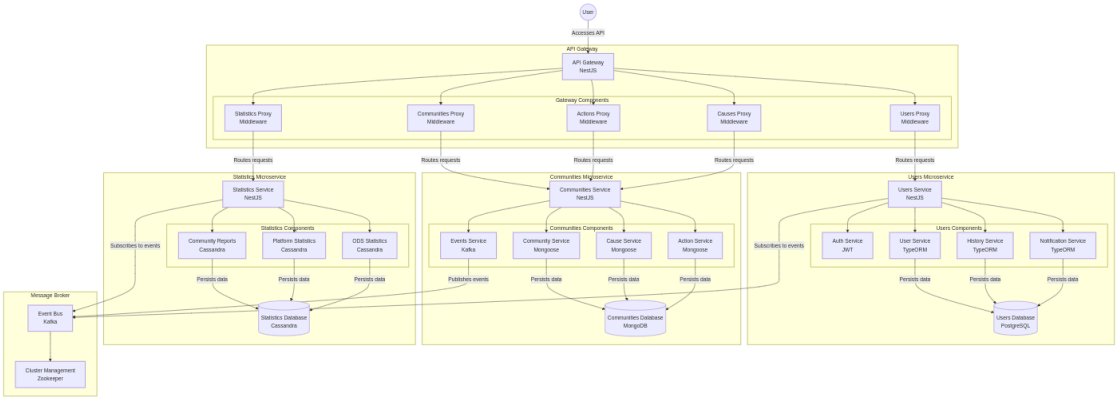

Figura 4. Diagrama de Arquitectura del Backend.

4.2 Explicación de los componentes y su integración.

La arquitectura de SolidarianID está basada en un enfoque de microservicios distribuidos, donde cada componente tiene una responsabilidad específica y se comunica de manera asíncrona a través de un broker de mensajería. El sistema cuenta con un API Gateway como punto de entrada único que gestiona las solicitudes de los clientes, autenticando y redirigiendo las peticiones al microservicio correspondiente. Este gateway incluye proxys dedicados que canalizan las solicitudes hacia los microservicios de Usuarios, Comunidades y Estadísticas.

El microservicio de Usuarios se encarga de gestionar la información de los perfiles de los usuarios, la autenticación mediante JWT, el historial de usuario y las notificaciones. Además, consume eventos en Kafka relacionados con la creación de comunidades, causas, acciones o contribuciones para actualizar su historial.

Por su parte, el microservicio de Comunidades administra la creación y gestión de comunidades, causas, acciones solidarias y contribuciones. El microservicio publica eventos en Kafka para notificar a otros servicios y asegurar la sincronización de la información en tiempo real.

El microservicio de Estadísticas recopila datos y genera reportes detallados sobre comunidades y estadísticas generales de la plataforma. Este microservicio consume los eventos publicados por el microservicio de Comunidades para actualizar continuamente sus estadísticas.

La integración entre los microservicios se realiza de manera asíncrona mediante Kafka, que actúa como un bus de eventos centralizado. Cada microservicio publica o consume eventos a través de tópicos dedicados, lo que permite mantenerlos desacoplados y garantizar la escalabilidad del sistema. Además, al utilizar bases de datos específicas para cada microservicio, se optimiza la persistencia de los datos según las necesidades particulares de cada uno.

5 Implementación de la arquitectura.

5.1 Descripción de la implementación de cada sistema de base de datos.

En la implementación de los microservicios, se ha adoptado el Patrón Repositorio para gestionar la interacción con las bases de datos. De esta manera conseguimos desacoplar la lógica de acceso a datos del resto de la aplicación, promoviendo modularidad, reutilización y facilidad de mantenimiento.

Para garantizar la consistencia entre los microservicios y sus diferentes bases de datos, se ha definido una clase abstracta genérica en una librería común compartida entre los servicios. Esta clase establece la interfaz base con las operaciones comunes a todos los repositorios, como guardado, búsqueda por ID y actualización (CRU), siguiendo el enfoque constructivo especificado para el MVP.

```
export abstract class Repository<T> {
  abstract save(entity: T): Promise<T>;
  abstract findById(id: string): Promise<T>;
}
```

Cada microservicio extiende esta clase genérica y proporciona la implementación específica de sus métodos, adaptándose a las particularidades de la base de datos que utiliza. En los siguientes apartados, se detalla la implementación concreta del patrón repositorio para cada sistema de base de datos.

5.1.1 Implementación para PostgreSQL.

Como se ha explicado, el sistema de PostgreSQL es el utilizado en el microservicio Usuarios, que se encarga de almacenar los usuarios, sus historiales y notificaciones. Para implementar el sistema de datos, en primer lugar tenemos la clase *UserRepository*, que extiende de la interfaz común *Repository* para la entidad User, definiendo el método *findByEmail.*

```
export abstract class UserRepository extends Repository<Domain.User> {
  abstract findByEmail(email: string): Promise<Domain.User>;
}
```

La implementación de este repositorio se encuentra en el paquete de infra/persistence. Para no contaminar el servicio de usuarios y el dominio con detalles de infraestructura, esta es la clase que se inyecta cuando se utiliza el repositorio *UserRepository*. La implementación del repositorio se realiza con TypeOrm, permitiendo utilizar sus funciones de *save*, *findOne* y *findOneBy.*

```
@Injectable()
export class UserRepositoryTypeOrm extends UserRepository {
  constructor(
    @InjectRepository(Persistence.User)
    private readonly userRepository: TypeOrmRepository<Persistence.User>,
  ) {
    super();
  }
```

Las funciones definidas para implementar el repositorio reciben y devuelven objetos del dominio, pero a la hora de almacenarlos o recuperarlos de la base de datos, utilizan la clase *UserMapper* para mapearlos de/a entidades de la persistencia, es decir, las que están anotadas con TypeOrm para la base de datos. A continuación se muestra un fragmento de la entidad User anotada para la persistencia:

```
@Entity()
export class User {
  @PrimaryColumn('uuid')
  id: string;
  @Column()
```

```
  firstName: string;
  @ManyToMany(() => User)
  @JoinTable({
    name: 'user_followers',
    joinColumn: { name: 'followed_id', referencedColumnName: 'id' },
    inverseJoinColumn: { name: 'follower_id', referencedColumnName: 'id' },
  })
  followers: User[];
```

Los módulos encargados de la gestión del historial (*history*) y notificaciones (*notifications*), que también pertenecen al microservicio usuarios, implementan el sistema de base de datos de la misma manera. Ambos definen una interfaz repositorio que extiende de la base y define los métodos específicos para la clase correspondiente y la implementan utilizando TypeOrm.

5.1.2 Implementación para MongoDB.

Para el microservicio comunidades, responsable de gestionar las comunidades de la aplicación, sus causas y acciones solidarias, se utilizó MongoDB. Se definen los repositorios para: acciones, causas, comunidades, solicitudes de creación de comunidad y solicitudes de unión a una comunidad. Todos ellos siguen la estructura explicada en el servicio usuarios, cada módulo define sus interfaces repositorio, extendiendo la de la librería común y luego las implementa con una clase *Injectable* para utilizar en los servicios. Se muestra un ejemplo de la interfaz definida para la entidad *Cause*:

```
export abstract class CauseRepository extends Repository<Domain.Cause> {
  abstract findAll(
    filter: CauseFilter,
    sort: CauseSort,
    pagination: PaginationParams,
  ): Promise<Domain.Cause[]>;
  abstract countDocuments(filter: CauseFilter): Promise<number>;
}
```

Para la implementación de los repositorios con MongoDB, se ha utilizado el modelo de ***mongoose***. Mongoose, al igual que TypeOrm en el caso de PostgreSQL, ofrece una capa de abstracción que permite interactuar con la base de datos de forma sencilla. Proporciona características para definir esquemas, realizar validaciones y trabajar con documentos JSON almacenados en MongoDB. Como ejemplo se muestra la implementación del repositorio de acciones:

```
@Injectable()
export class ActionRepositoryMongoDB extends ActionRepository {
  constructor(
    @InjectModel(Persistence.Action.name)
    private readonly actionModel: Model<Persistence.Action>,
  ) {
    super();
  }
```

Aquí se inyecta el modelo de Mongoose correspondiente a la entidad Action (*ActionModel*), utilizando el decorador proporcionado por NestJS, y se utiliza para realizar las operaciones del repositorio. Al igual que en el modelo de PostgreSQL para usuarios, este repositorio es el encargado de mapear las entidades del dominio a la persistencia y viceversa.

Para anotar las clases con mongoose, utilizamos el decorador de *Schema* para la clase y el de *Prop* para sus atributos. Utilizamos el *SchemaFactory* de mongoose para exportar el esquema de la clase. Se muestra un fragmento de la clase *Action* anotada:

```
@Schema({ timestamps: true }) automatically
export class Action {
  @Prop({ required: true, unique: true })
  id: string;

  @Prop({ required: true, type: String, enum: ActionStatus })
  status: ActionStatus;

  @Prop({ required: true, type: String, enum: ActionType })
  type: ActionType;

  [...]
}
export const ActionSchema = SchemaFactory.createForClass(Action);
```

Esta estructura se sigue para todos los repositorios utilizados en el microservicio comunidades, definiendo los esquemas e implementando los repositorios con mongoose.

5.1.2.1 Implementación de filtros y paginación.

En el sistema implementado para MongoDB, cabe destacar el uso de filtros y ordenación. En este microservicio se necesitaba implementar varias consultas que requerían filtrar, ordenar y paginar los resultados, como las de obtener el listado de causas solidarias, filtrando por ODS o la de obtener el listado de acciones solidarias filtrando por estado.

Para implementar estos filtros y paginación de manera eficiente, en el repositorio se hace uso de los métodos *find, sort, skip,* y *limit* del modelo de mongoose. Estas funciones permiten manejar el orden y paginación de manera sencilla. En el siguiente fragmento se muestra su uso para devolver el listado filtrado, ordenado y paginado, según los parámetros recibidos, de las causas solidarias almacenadas. Además, se implementó la función *countDocuments*, que también recibe los filtros, para contar el total de causas que cumplen dicho filtro.

```
async findAll(
    filter: CauseFilter,
    sort: CauseSort,
    pagination: PaginationParams,
  ): Promise<Domain.Cause[]> {
    const causes = await this.causeModel
      .find(filter)
      .sort(sort)
      .skip(pagination.skip)
      .limit(pagination.limit)
      .exec();
    return causes.map(CauseMapper.toDomain);
 }
 async countDocuments(filter: CauseFilter): Promise<number> {
    return this.causeModel.countDocuments(filter).exec();
 }
```

En cuanto a los parámetros necesarios para aplicar estos filtros en el modelo de mongoose, se han definido varias estructuras. Se detalla la implementación de estas para el ejemplo del módulo causas, se han realizado de manera análoga para los de comunidades y acciones.

En primer lugar, en la librería común se definen los enumerados:

- ***CauseSortBy***: sus miembros son *TITLE* y *CREATED_AT*, para permitir ordenar las causas por título y por fecha de creación

- ***SortDirection***: sus miembros son *asc* y *desc*, para poder ordenar los listados de manera ascendente o descendente

- ***PaginationDefaults***: incluye las opciones por defecto para la paginación (página 1, límite 10 y límite máximo 100).

Por otro lado, en el fichero 'causes/infra/filters/cause-query.builder.ts' se definen los tipos:

- ***CauseFilter***: define los filtros para las consultas de causas en la base de datos. Contiene dos propiedades opcionales: *ods*, que utiliza el operador *$in* para buscar registros cuyo valor de *ods* esté dentro de un conjunto de números específicos (un array de IDs de los ods), y *title*, que utiliza los operadores *$regex* y *$options* para realizar una búsqueda de texto en el campo *title* utilizando una expresión regular, permitiendo realizar búsquedas parciales y personalizadas, como búsquedas insensibles a mayúsculas o minúsculas.

- ***CauseSort***: se utiliza para especificar el orden de los resultados de la consulta a la base de datos (1 indica un orden ascendente, y -1 un orden descendente).

- ***PaginationParams***: para especificar los parámetros de paginación en una consulta. Tiene dos parámetros: *skip*, el número de registros que se deben omitir al inicio de la consulta, y *limit*, el número máximo de registros que se deben devolver en la consulta.

```
export type CauseFilter = {
  ods?: { $in: number[] };
  title?: { $regex: string; $options: string };
};

export type CauseSort = Record<string, 1 | -1>;

export type PaginationParams = {
  skip: number;
  limit: number;
};
```

También en este fichero, se define la clase *CauseQueryBuilder*, que se encarga de inicializar y gestionar todos estos parámetros de ordenación, paginación y filtros. Como se observa en el ejemplo del método *addSort*, para inicializar los parámetros de ordenación, se hace uso de los enumerados comentados anteriormente *CauseSortBy* y *SortDirection*:

```
  addSort(
    sortBy?: CauseSortBy,
    sortDirection: SortDirection = SortDirection.ASC,
  ): this {
    if (sortBy) {
      this.sort[sortBy] = sortDirection === SortDirection.DESC ? -1 : 1;
    }
    return this;
  }
```

Esta clase se utiliza en el servicio de causas para recoger los parámetros necesarios y trasladarlos al repositorio, responsable de realizar la búsqueda correspondiente. Aquí se observa cómo se utiliza el *query builder* en el servicio:

```
const queryBuilder = new CauseQueryBuilder()
      .addOdsFilter(odsFilter)
      .addNameFilter(nameFilter)
```

```
      .addSort(sortBy, sortDirection)
      .addPagination(page, limit);

    const filters = queryBuilder.buildFilter();
    const sort = queryBuilder.buildSort();
    const pagination = queryBuilder.buildPagination();

    // Use Promise.all to execute both queries in parallel
    const [data, total] = await Promise.all([
      this.causeRepository.findAll(filters, sort, pagination),
      this.causeRepository.countDocuments(filters),
    ]);
```

5.1.3 Implementación para Cassandra.

Por último, el sistema Cassandra se utiliza en el microservicio de estadísticas para almacenar datos analíticos y reportes de las comunidades. La implementación de este sistema combina la definición de un esquema optimizado para consultas de lectura y escritura rápidas con el uso del patrón repositorio que sigue la estructura general adoptada en todos los microservicios.

En este caso, la implementación varía un poco respecto a las de PostgreSQL y MongoDB, debido a las peculiaridades de esta base de datos. Por una parte, definimos el esquema 'statistics-ms/cassandra_schema.cql', en el que se diseñan las tablas y estructuras de datos necesarias, incluyendo claves de partición y clustering, utilizando CQL (Cassandra Query Language). A continuación se describe cómo se realiza la implementación para el caso de las estadísticas de una comunidad por ods, las demás clases de este microservicio se implementan de la misma forma..

Para gestionar las operaciones de lectura y escritura, se implementó un repositorio específico para cada entidad, utilizando la librería **cassandra-for-nest.** Esta librería facilita la interacción con Cassandra en aplicaciones NestJS, proporcionando decoradores *@InjectClient* y *@InjectMapper* para inyectar el cliente y el mapeador de Cassandra, respectivamente. El repositorio extiende de *BaseService* de cassandra-for-nest, lo que permite aprovechar métodos predefinidos para operaciones comunes. Al igual que con los demás sistemas implementados, el repositorio específico para Cassandra trabaja con objetos del dominio y utiliza el mapper específico *OdsCommunityMapper* para mapear las entidades del dominio a la persistencia y viceversa.

```
@Injectable()
export default class OdsCommunityRepository extends
BaseService<Persistence.OdsCommunity> {
  constructor(
    @InjectClient() client: Client,
    @InjectMapper(Persistence.OdsCommunity) mapper: mapping.Maper,
  ) {
    super(client, mapper, Persistence.OdsCommunity);
  }
```

Las entidades se definen utilizando decoradores como *@Entity* y *@Column*, de *cassandra-for-nest* indicando el nombre de la tabla y las columnas correspondientes. Esta definición facilita el mapeo entre las entidades del dominio y las estructuras de persistencia.

```
@Entity({
  keyspace: envs.cassandraKeyspace,
  table: 'ods_community',
})
export class OdsCommunity {
  @Column({ name: 'ods_id' })
  odsId: number; // ODS Identifier

  @Column({ name: 'community_id' })
```

```
  communityId: string; // Community Identifier
}
```

5.2 Consistencia de datos

La consistencia de datos se refiere al principio por el cual los datos en un sistema deben permanecer correctos, coherentes y válidos según las reglas y restricciones definidas para ellos, independientemente de las operaciones realizadas o los fallos que puedan ocurrir.

La consistencia de datos se aplica de manera diferente en cada microservicio, dependiendo del tipo de base de datos utilizada y de las necesidades específicas del dominio de cada servicio. A continuación, se explica cómo se implementa la consistencia en cada uno de los microservicios.

En el microservicio de Usuarios, que utiliza PostgreSQL, la consistencia se asegura mediante las propiedades *ACID*, garantizando que todas las transacciones sean completas, válidas y coherentes. Esto incluye el uso de claves foráneas para mantener la integridad referencial entre entidades como perfiles, historial de actividades y relaciones de usuarios. Las transacciones permiten que las operaciones sean revertidas si fallan, evitando datos inconsistentes. Además, constraints como restricciones únicas y validaciones personalizadas aseguran que los datos almacenados cumplan con las reglas del negocio en todo momento.

En el microservicio de Comunidades, que utiliza MongoDB, la consistencia se adapta según las necesidades específicas de las operaciones. Para operaciones críticas, como la creación de relaciones entre comunidades y acciones, MongoDB permite transacciones que garantizan que todas las operaciones relacionadas se completen de manera coherente. En operaciones menos críticas, se emplea una consistencia eventual, optimizando la velocidad de escritura y lectura. Además, el nivel de confirmación de escritura (*write concern*) puede ajustarse para garantizar la persistencia en réplicas antes de finalizar la operación, balanceando rendimiento y fiabilidad.

El microservicio de Estadísticas, que utiliza Cassandra, implementa un modelo de consistencia eventual para manejar grandes volúmenes de datos provenientes de Kafka. Los datos son replicados en múltiples nodos, lo que garantiza alta disponibilidad incluso en caso de fallos. La consistencia se ajusta mediante configuraciones como *QUORUM*, que asegura que una mayoría de nodos confirmen las operaciones antes de considerarlas válidas. Esto permite un equilibrio entre rendimiento y precisión, siendo ideal para escrituras rápidas y consultas eficientes de datos históricos y series temporales.

5.3 Replicación y particionamiento.

La replicación consiste en crear copias de los datos en diferentes nodos para garantizar alta disponibilidad, tolerancia a fallos y balanceo de carga en las lecturas. Por otro lado, el particionamiento se refiere a dividir los datos en subconjuntos más pequeños (particiones o shards) distribuidos entre múltiples nodos, lo que permite manejar grandes volúmenes de información y mejorar el rendimiento al distribuir la carga de trabajo.

Lamentablemente, no se dispuso del tiempo necesario para implementar esta parte en el desarrollo actual del sistema. Sin embargo, a continuación se describen las estrategias y pasos que podrían llevarse a cabo para su implementación en el futuro.

5.3.1 Microservicio de Usuarios

En el microservicio de usuarios, para la replicación podemos configurar la replicación *maestro-esclavo* mediante el sistema de streaming replicación navito de PostgreSQL. El nodo

maestro gestiona las escrituras, mientras que los esclavos manejan las lecturas. Si el maestro falla, un esclavo puede ser promovido a maestro usando herramientas como *Patroni*.

Para la parte de particionamiento podemos implementar particionamiento por rango utilizando fechas de creación como criterio para dividir los datos en tablas más pequeñas (por ejemplo, usuarios creados por mes). Esto mejora la velocidad de acceso y reduce el tamaño de las consultas a tablas grandes.

5.3.2 Microservicio de Comunidades

Para el microservicio de comunidades podemos implementar un *Replica Set* para la parte de replicación. Se consigue con un nodo primario que gestione las escrituras y varios secundarios que manejen las lecturas y sirvan como respaldo. Configurar un nodo árbitro para garantizar elecciones rápidas en caso de fallos del primario.

Para la parte de particionamiento podemos usar *sharing* con la clave de partición basada en IDs de comunidad. Esto distribuye los datos de manera uniforme entre los shards y asegura que las consultas relacionadas con una comunidad específica sean rápidas y eficientes.

5.3.2 Microservicio de Estadísticas

En el microservicio de estadísticas, para la parte de replicación podemos configurar un *Replication Factor* de 3 para que cada dato esté almacenado en tres nodos diferentes. Esto garantiza tolerancia a fallos y disponibilidad, incluso si dos nodos fallan. Además, configurar la replicación entre centros de datos (multi-datacenter) para conseguir una alta disponibilidad global.

Para la parte de particionamiento podríamos usar una clave de partición basada en *timestamps* para distribuir los datos de eventos y métricas entre los nodos. Esto asegura un balance uniforme de la carga y optimiza la escritura masiva que realiza este microservicio.

5.4 Escalabilidad y tolerancia a fallos.

Por la falta de tiempo no hemos podido llevar a cabo la implementación de esta parte, pero en caso de poder implementar la escalabilidad y la tolerancia a fallos en las bases de datos de cada microservicio, se deben emplear diversas estrategias adaptadas a las características de cada base de datos.

En el microservicio de usuarios, que utiliza PostgreSQL, la escalabilidad se puede lograr mediante replicación en modo maestro-esclavo o maestro-maestro, particionamiento de datos y el uso de herramientas como *Citus* para convertir PostgreSQL en un clúster distribuido. La tolerancia a fallos se logra mediante configuraciones de failover automático con herramientas como *Patroni*, un sistema de backups regulares y balanceo de carga entre réplicas para garantizar alta disponibilidad.

En el microservicio de Estadísticas, que utiliza Cassandra, la escalabilidad se gestiona fácilmente mediante su arquitectura distribuida que permite agregar nodos al clúster de manera horizontal sin afectar el rendimiento. Además, el particionado de datos y el uso de replicación multi-datacenter aseguran alta disponibilidad y recuperación en caso de fallos. La tolerancia a fallos en Cassandra se optimiza mediante ajustes de consistencia y herramientas de monitoreo como *Prometheus* y *Grafana* para una recuperación rápida de nodos fallidos.

En el Microservicio de Comunidades, que utiliza MongoDB, la escalabilidad se logra con sharding para distribuir los datos entre múltiples nodos y la creación de índices adecuados para mejorar el rendimiento de las consultas. La tolerancia a fallos se asegura mediante *Replica*

Sets, que permiten la replicación automática de datos y el failover en caso de caídas del nodo principal, junto con un monitoreo constante mediante herramientas como *MongoDB Atlas*.

6 Plataforma de procesamiento de datos. Apache Kafka.

En SolidarianID, el uso de Kafka como plataforma de procesamiento de datos permite gestionar de manera eficiente la comunicación entre microservicios, asegurando escalabilidad y un alto grado de desacoplamiento. A través de Kafka, los Domain Events, como *UserJoinedCommunity* o *ActionCreatedEvent*, se publican en tópicos específicos utilizando el *KafkaEventPublisherService*. Estos tópicos actúan como canales centralizados donde otros microservicios, como el de Usuarios o Estadísticas, pueden suscribirse para consumir los eventos relevantes y reaccionar en tiempo real, ya sea actualizando métricas, registros de historial o cualquier lógica asociada.

Kafka destaca por su capacidad de manejar flujos de datos masivos y en tiempo real con baja latencia, lo que es clave para garantizar un rendimiento óptimo incluso bajo alta carga. Además, su diseño basado en la persistencia de mensajes asegura que los eventos se conserven en los tópicos hasta que los consumidores los procesen, lo que aporta resiliencia al sistema al permitir que los microservicios puedan recuperarse de fallos o interrupciones sin pérdida de datos.

Otra ventaja clave de Kafka es su soporte para patrones avanzados como *Event Sourcing*. En SolidarianID, estos patrones permiten que los cambios en el estado del sistema se registren como eventos almacenados en los tópicos de Kafka, lo que no solo facilita la sincronización en tiempo real, sino también la trazabilidad, auditorías y análisis históricos de los datos.

Para facilitar y abstraer el sistema de comunicación entre microservicios, Kafka se ha implementado como un módulo de configuración compartido. Esta estrategia simplifica la integración de nuevos componentes, promueve la reutilización y garantiza uniformidad en la configuración de los diferentes microservicios de la plataforma.

En conjunto, Kafka desempeñó un papel central en la arquitectura de este proyecto al ofrecer un sistema de mensajería robusto y eficiente. Su capacidad para manejar flujos masivos de datos en tiempo real, combinada con su diseño orientado a eventos, asegura que la plataforma esté preparada para evolucionar de manera escalable y adaptarse a las posibles futuras necesidades de la aplicación.

7 Consultas y procesamiento de datos.

En este apartado se analizarán las consultas más relevantes utilizadas en los casos de uso principales del backend de cada microservicio, destacando cómo cada una optimiza o podría mejorar el rendimiento del sistema. Las consultas de guardado que son similares entre sí se omitirán en este documento para evitar redundancias.

7.1 Microservicio de Usuarios.

UserRepositoryTypeOrm.

1. **Búsqueda de Usuario por id (findById):**

```
const user = await this.userRepository.findOne({
```

```
  where: { id },
  relations: ['followers'],
});
```

En esta consulta se utiliza *relations* para precargar la lista de seguidores en una sola consulta, evitando múltiples consultas separadas y mejorando así la eficiencia en accesos repetidos. Se asume que, para la versión MVP, los usuarios no tendrán un número muy elevado de seguidores/seguidos.

Para futuras versiones, sería recomendable implementar lazy loading para cargar la lista de seguidores únicamente cuando se necesite mostrarla en el perfil del usuario, en lugar de cargarla en cada consulta relacionada al usuario. Esto optimizaría el rendimiento al reducir la cantidad de datos cargados innecesariamente.

La consulta realiza la búsqueda utilizando id, el cual está indexado por ser *@PrimaryColumn*, garantizando una operación de acceso rápida y eficiente.

2. **Búsqueda de Usuario por email (findByEmail):**

```
const user = await this.userRepository.findOneBy({ email });
```

La columna *email* está marcada como única, lo que crea automáticamente un índice. Además, se añade *@Index* para hacer explícito que se busca optimizar las consultas en esa columna. Esto garantiza que la búsqueda se realice de manera eficiente mediante índices.

HistoryEntryRepositoryTypeorm.

1. **Historial de Actividades de un Usuario con Filtros (findByUserIdWithFilters):**

```
const query = this.historyEntryRepository
  .createQueryBuilder('history_entry')
  .where('history_entry.userId = :userId', { userId })
  .andWhere('history_entry.type = :type', { type })
  .andWhere('history_entry.status = :status', { status })
  .orderBy('history_entry.timestamp', 'DESC')
  .skip((page - 1) * limit)
  .take(limit)
  .getMany();
```

La optimización de la consulta se logra mediante el uso de un índice compuesto que incluye las columnas *userId*, *type*, *status* y *timestamp*. Este índice permite filtrar eficientemente por *userId*, *type* y *status*, y ordenar los resultados por timestamp sin realizar un escaneo completo de la tabla. Además, se implementa paginación mediante *skip* y *take*, evitando la carga de grandes volúmenes de datos en una sola solicitud. La ordenación descendente por *timestamp* garantiza que las actividades más recientes se presenten primero, mejorando la relevancia de los resultados mostrados.

2. **Conteo de Actividades (countByUserIdWithFilters):**

```
const query = this.historyEntryRepository
  .createQueryBuilder('history_entry')
  .where('history_entry.userId = :userId', { userId })
  .andWhere('history_entry.type = :type', { type })
  .andWhere('history_entry.status = :status', { status })
  .getCount();
```

La optimización de esta consulta también se beneficia del índice compuesto *('userId', 'type', 'status', 'timestamp')*, ya que permite contar los registros filtrados sin necesidad de cargar todas las filas completas en memoria. Al usar *getCount()*, se obtiene directamente el número total de actividades que coinciden con los filtros, reduciendo el uso de recursos y mejorando el rendimiento en respuestas rápidas.

NotificationRepositoryTypeorm.

1. Notificaciones de un Usuario (findByUserId):

```
const notifications = await this.notificationRepository.find({
  where: { userId },
  skip: (page - 1) * limit,
  take: limit,
  order: { timestamp: 'DESC' },
});
```

La consulta optimiza los resultados al filtrar directamente por *userId*, el cual cuenta con un índice para limitar el número de filas escaneadas y mejorar el rendimiento de búsqueda. Asimismo, la ordenación por *timestamp*, también indexado, garantiza que las notificaciones se ordenen de manera eficiente, mostrando primero las más recientes. Además, se implementa paginación para evitar la carga de consultas grandes.

2. Marcar Notificación como Leída (markAsRead):

```
const notification = await this.notificationRepository.findOne({
  where: { id: notificationId, userId },
});
notification.read = true;
await this.notificationRepository.save(notification);
```

La consulta realiza una búsqueda directa utilizando las claves únicas *id* y *userId*, lo que permite localizar la notificación de manera rápida y eficiente. Además, la transacción es simple, ya que solo se lleva a cabo una actualización parcial del campo *read = true*, evitando operaciones complejas sobre múltiples campos y optimizando el rendimiento de la operación.

7.2 Microservicio de Comunidades.

CommunityRepositoryMongoDb.

1. **Buscar Comunidad por Nombre (findByName):**

```
const existsCommunity = await this.communityModel.findOne({ name });
```

La consulta se optimiza creando un índice en el campo *name*, lo que permite a MongoDB localizar los documentos directamente sin escanear toda la colección, lo que reduce significativamente el tiempo de búsqueda.

2. **Verificar si un Usuario es Admin de una Comunidad (isCommunityAdmin):**

```
const existsCommunity = await this.communityModel.findOne({
  adminId: userId,
  id: communityId,
});

CommunitySchema.index({ adminId: 1, id: 1 });
```

La consulta se optimiza creando un índice compuesto en los campos *adminId* e *id*, lo que permite a MongoDB localizar rápidamente los documentos que coinciden con ambos criterios sin realizar un escaneo completo de la colección. Este índice mejora significativamente el rendimiento al agilizar la búsqueda de comunidades filtradas por el administrador y el identificador de la comunidad.

3. **Buscar Comunidad por ID (findById):**

```
const existsCommunity = await this.communityModel.findOne({ id });
```

MongoDB crea automáticamente un índice en la clave primaria *_id*. Sin embargo, en nuestro caso el campo *id* es distinto de *_id*, es importante contar con un índice único para garantizar búsquedas rápidas y asegurar la integridad de los datos, evitando duplicados en este campo.

4. **Guardar o Actualizar una Comunidad (save):**

```
await this.communityModel.findOneAndUpdate(
  { id: entity.id.toString() },
  document,
  { upsert: true, new: true },
).exec();
```

La consulta se optimiza utilizando *findOneAndUpdate* con *upsert: true*, lo que permite combinar la búsqueda, creación o actualización en una única operación. Esto evita múltiples consultas al servidor de MongoDB, reduciendo la latencia y mejorando el rendimiento. Además, el campo *id* tiene un índice único, por lo que la operación es aún más eficiente, ya

que MongoDB puede localizar rápidamente el documento correspondiente o insertar uno nuevo si no existe.

5. **Listar Comunidades con Filtros y Paginación (findAll):**

```
const communities = await this.communityModel
  .find(filter)
  .skip(pagination.skip)
  .limit(pagination.limit)
  .exec();
```

La consulta utiliza *filter* para reducir la cantidad de documentos escaneados y emplea *skip* y *limit* para implementar paginación, evitando que se cargue toda la colección en memoria. Además, el filtro empleado es por el campo nombre para el cual ya hemos indicado que tiene un índice que optimiza la búsqueda.

6. **Contar Comunidades con Filtros (countDocuments):**

```
const count = await this.communityModel.countDocuments(filter).exec();
```

Al igual que en la consulta anterior, el filtro por nombre se aplica sobre un campo indexado.

CreateCommunityRequestRepositoryMongoDb.

1. **Buscar Solicitud por ID (findById):**

```
const existsRequest = await this.createCommunityModel.findOne({ id });
```

Igual que en CommunityRepositoryMongoDb.

2. **Guardar o Actualizar Solicitud (save):**

```
await this.createCommunityModel.findOneAndUpdate(
  { id: entity.id.toString() },
  document,
  { upsert: true, new: true },
).exec();
```

Igual que en CommunityRepositoryMongoDb.

3. **Listar Solicitudes con Filtros y Paginación (findAll):**

```
const requests = await this.createCommunityModel
  .find(filter)
  .sort(sort)
  .skip(pagination.skip)
```

```
  .limit(pagination.limit)
  .exec();

CreateCommunityRequestSchema.index({ status: 1, createdAt: -1 });
```

La consulta se optimiza gracias al índice compuesto en los campos *status* y *createdAt*, lo que permite a MongoDB localizar rápidamente los documentos que coinciden con ambos criterios sin realizar un escaneo completo de la colección y ordenarlos de manera eficiente por los más recientes.

4. **Contar Documentos con Filtros (countDocuments):**

```
const count = await this.createCommunityModel.countDocuments(filter).exec();
```

Al igual que en la consulta anterior, el filtro por se aplica sobre campos indexados.

JoinCommunityRequestRepositoryMongoDb.

1. **Buscar Solicitud por ID (findById):**

```
const existsRequest = await this.joinCommunityModel.findOne({ id });
```

Igual que en consultas anteriores por *id*.

2. **Guardar o Actualizar Solicitud (save):**

```
await this.joinCommunityModel.findOneAndUpdate(
  { id: entity.id.toString() },
  document,
  { upsert: true, new: true },
).exec();
```

Igual que en consultas anteriores.

3. **Listar Solicitudes Pendientes con Paginación (findAll):**

```
const requests = await this.joinCommunityModel
  .find({ status: StatusRequest.PENDING, communityId })
  .skip(pagination.skip)
  .limit(pagination.limit)
  .exec();

JoinCommunityRequestSchema.index({ communityId: 1, status: 1 });
```

La consulta se optimiza gracias al índice compuesto en los campos *communityId* y *status*, lo que permite a MongoDB localizar rápidamente los documentos que coinciden con ambos criterios sin realizar un escaneo completo de la colección.

4. **Contar Solicitudes por Comunidad (countDocuments):**

```
const count = await this.joinCommunityModel.countDocuments
({ communityId }).exec();
```

Se crea como en anteriores consultas un índice simple para *communityId*.

5. **Solicitud por Usuario y Comunidad (findByUserIdAndCommunityId):**

```
const existsRequest = await this.joinCommunityModel.findOne({
  userId,
  communityId,
});

JoinCommunityRequestSchema.index({ userId: 1, communityId: 1 });
```

La consulta se optimiza gracias al índice compuesto en los campos *userId* y *communityId*, lo que permite a MongoDB localizar rápidamente los documentos que coinciden con ambos criterios sin realizar un escaneo completo de la colección.

CauseRepositoryMongoDB.

1. **Guardar o Actualizar una Causa (save):**

```
const doc = await this.causeModel
  .findOneAndUpdate(
    { id: persistenceCause.id },
    persistenceCause,
    { upsert: true, new: true },
  )
  .exec();
```

Se utiliza *findOneAndUpdate* con *upsert: true* para combinar la búsqueda, creación o actualización del documento en una sola operación. La búsqueda se realiza mediante el campo *id*, que cuenta con un índice simple, lo que permite localizar el documento de manera eficiente sin escanear toda la colección.

2. **Buscar Causa por ID (findById):**

```
const cause = await this.causeModel.findOne({ id }).exec();
```

Igual que las anteriores. Se filtra sobre un campo indexado.

3. **Listar Causas con Filtros y Paginación (findAll):**

```
const causes = await this.causeModel
  .find(filter)
```

```
  .sort(sort)
  .skip(pagination.skip)
  .limit(pagination.limit)
  .exec();

CauseSchema.index({ ods: 1, title: 1, createdAt: -1 });
```

La consulta se optimiza gracias al índice compuesto en los campos *ods*, *title* y *createdAt*, lo que permite a MongoDB localizar rápidamente los documentos que coinciden con los criterios de búsqueda sin realizar un escaneo completo de la colección y ordenarlos eficientemente por los más recientes. Además, el uso de *skip* y *limit* evita cargar todos los documentos a la vez, devolviendo solo una página de resultados. Dado que los valores de *skip* no son elevados, no se considera necesaria una paginación basada en cursores.

4. **Contar Documentos con Filtros (countDocuments):**

```
const count = await this.causeModel.countDocuments(filter).exec();
```

Igual que en consultas anteriores, los campos por los que se filtran son parte de un índice compuesto.

ActionRepositoryMongoDB.

1. **Guardar o Actualizar una Acción (save):**

```
const doc = await this.actionModel
  .findOneAndUpdate({ id: persistenceAction.id }, persistenceAction, {
    upsert: true,
    new: true,
  })
  .exec();
```

Se utiliza *findOneAndUpdate* con *upsert: true* para combinar la búsqueda, creación o actualización del documento en una sola operación. La búsqueda se realiza mediante el campo *id*, que cuenta con un índice simple, lo que permite localizar el documento de manera eficiente sin escanear toda la colección.

2. **Buscar Acción por ID (findById):**

```
const action = await this.actionModel.findOne({ id }).exec();
```

Igual que en consultas anteriores.

3. **Listar Acciones con Filtros y Paginación (findAll):**

```
const actions = await this.actionModel
  .find(filter)
  .sort(sort)
```

```
  .skip(pagination.skip)
  .limit(pagination.limit)
  .exec();

ActionSchema.index({ status: 1, title: 1, createdAt: -1, type: 1 });
```

La consulta se optimiza gracias al índice compuesto en los campos *status*, *title*, *createdAt* y *type*, lo que permite a MongoDB localizar de manera eficiente los documentos que coinciden con los criterios de búsqueda sin realizar un escaneo completo de la colección y ordenarlos según cualquiera de estos campos. El uso de *skip* y *limit* evita cargar todos los documentos de una vez, devolviendo solo una página de resultados. Dado que los valores de *skip* son bajos, no se considera necesaria la implementación de una paginación basada en cursores.

4. **Contar Documentos con Filtros (countDocuments):**

```
const count = await this.actionModel.countDocuments(filter).exec();
```

Igual que en consultas interiores, se filtra por índices ya indexados.

5. **Buscar Acciones por causeId (findByCauseId):**

```
const actions = await this.actionModel.find({ causeId }).exec();
```

Se crea como en anteriores consultas un índice simple para *causeId*.

7.3 Microservicio de Estadísticas.

CommunityByCommunityIdRepository.

Este microservicio utiliza *BaseService* de *cassandra-for-nest*, que emplea internamente consultas preparadas para reducir la sobrecarga de compilación de consultas y mejorar los tiempos de ejecución, optimizando el rendimiento de las operaciones de base de datos.

En este microservicio, la paginación no se puede aplicar, ya que el frontend requiere todos los datos de una vez para generar los informes de estadísticas de manera completa.

1. **Buscar Comunidad por communityId (findOneByCommunityId):**

```
  const entity = await this.findOne({ communityId });
```

La búsqueda utiliza *communityId* que es clave primaria en el esquema, lo que garantiza una consulta eficiente al acceder directamente a la partición correspondiente sin necesidad de escaneos, optimizando el rendimiento por diseño.

2. **Guardar o Actualizar una Comunidad (save):**

```
const entity = CommunityByCommunityIdMapper.toPersistence(community);
```

```
return this.saveOne(entity);
```

El método *saveOne* realiza un *INSERT* que sobrescribe automáticamente los datos si la clave primaria ya existe, evitando la necesidad de una consulta previa con *findOne*. Esto optimiza el rendimiento al hacer las inserciones idempotentes y reducir el número de operaciones en la base de datos.

CauseByCommunityIdRepository

1. **Listar Causas por communityId (findManyByCommunityId):**

```
const entity = await this.findMany({ communityId });
return entity.map(CauseByCommunityIdMapper.toDomain);
```

La búsqueda por *communityId* es eficiente, ya que Cassandra accede a una única partición al ser este campo la clave de partición. Además, los resultados se devuelven ordenados por *cause_id*, ya que este atributo se utiliza como clave de clustering en el esquema.

2. **Buscar Causa Específica por communityId y causeId (findOneByCauseId):**

```
const entity = await this.findOne({ communityId, causeId });
return CauseByCommunityIdMapper.toDomain(entity);
```

Esta consulta utiliza *community_id* como clave de partición y *cause_id* como clave de clustering, lo que permite acceder directamente a una fila específica. Al combinar ambos campos, la consulta es extremadamente rápida y no requiere escanear la tabla completa.

ActionByCauseIdRepository.

1. **Listar Acciones por causeId (findManyByCauseId):**

```
const entities = await this.findMany({ causeId });
return entities.map(ActionByCauseIdMapper.toDomain);
```

La consulta utiliza *causeId* como clave de partición, lo que garantiza el acceso directo a la partición correspondiente. Además, al ser *action_id* la clave de clustering, las acciones se devuelven automáticamente ordenadas por este campo, optimizando tanto la búsqueda como la ordenación.

2. **Buscar una Acción por causeId y actionId (findOneByActionId):**

```
const entity = await this.findOne({ causeId, actionId });
return ActionByCauseIdMapper.toDomain(entity);
```

La consulta utiliza *causeId* como clave de partición y *actionId* como clave de clustering, lo que permite acceder directamente a una fila específica sin recorrer toda la partición. Al

combinar ambos campos, Cassandra ejecuta la consulta de manera eficiente, evitando escaneos completos.

OdsStatisticsRepository.

1. **Obtener Todas las Estadísticas (findAllEntities):**

```
const results = await this.findAll();
return results.map(OdsStatisticsMapper.toDomain);
```

Al utilizar *findAll()*, Cassandra realiza un escaneo de todas las filas de la tabla, lo que generalmente no es eficiente en tablas con muchas filas, ya que Cassandra no está optimizado para escaneos completos. Sin embargo, en este caso, el número de filas está limitado a 17, correspondientes a los ODS disponibles, lo que hace que el impacto en el rendimiento sea mínimo.

2. **Buscar Estadísticas de un ODS por odsId (findOneEntity):**

```
const entity = await this.findOne({ odsId });
```

La consulta utiliza *odsId* como clave de partición, lo que permite acceder directamente a una única fila. Cassandra recupera los datos desde la partición correspondiente sin realizar escaneos completos, lo que hace que esta operación sea eficiente por diseño.

3. **Obtener el Total de Apoyos (getTotalSupports):**

```
const cql = `SELECT SUM(supports_count) AS total_supports FROM ${this.keyspaceName}.${this.tableName}`;
const result = await this.mapCqlAsExecution(cql, undefined, undefined)({});
const totalSupports = result.first()?.['total_supports'] || 0;
return totalSupports;
```

Esta consulta utiliza *SUM()* para agregar los valores de *supports_count*. Las funciones de agregación como *SUM()* requieren escanear todas las filas de la partición, lo que puede ser costoso. Una posible alternativa más eficiente sería usar una fila con un id que no pueda existir, como *-1*, para almacenar el valor agregado de la suma de manera precomputada. De esta forma, la consulta solo necesitaría recuperar esta fila específica, evitando escaneos y mejorando significativamente el rendimiento.

OdsCommunityRepository.

1. **Buscar una Comunidad por odsId y communityId (findOneEntity):**

```
const community = await this.findOne({ odsId, communityId });
```

La consulta utiliza *odsId* como clave de partición y *communityId* como clave de clustering, lo que permite acceder directamente a una fila específica. Cassandra utiliza esta clave primaria compuesta para localizar el registro de manera eficiente, sin necesidad de escanear toda la tabla.

CommunityStatisticsRepository.

1. **Buscar Estadísticas por communityId (findOneEntity):**

```
const entity = await this.findOne({ communityId });
```

La consulta utiliza *communityId* como clave de partición, lo que permite acceder directamente a una única fila.

2. **Obtener Todas las Estadísticas (findAllEntities):**

```
const entities = await this.findAll();
```

La consulta *findAll()* realiza un escaneo completo de la tabla *community_statistics*, lo que puede ser costoso si la tabla contiene muchas comunidades, ya que Cassandra no está optimizado para escanear grandes volúmenes de datos. Sin embargo, en este caso, es necesario recuperar todos los datos de una sola vez para elaborar los gráficos de estadísticas de manera completa, lo que justifica el uso de esta operación.

3. **Obtener el Total de Apoyos (getTotalSupports):**

```
const cql = `SELECT SUM(support_count) AS total_supports FROM ${this.keyspaceName}.${this.tableName}`;
const result = await this.mapCqlAsExecution(cql, undefined, undefined)({});
const totalSupports = result.first()?.['total_supports'] || 0;
return totalSupports;
```

Esta consulta ejecuta la función de agregación *SUM()* en todos los registros de la tabla. Como *SUM()* requiere escanear todas las filas, el rendimiento puede verse afectado si la tabla contiene una gran cantidad de datos. Una alternativa más eficiente es utilizar un índice no válido, como -1, para almacenar el resultado precomputado de la suma. De esta manera, en lugar de escanear la tabla, la consulta simplemente recupera el valor almacenado, mejorando significativamente el rendimiento.

8 Descripción de configuración del Docker e instrucciones despliegue.

El proyecto SolidarianID está disponible en el repositorio de GitHub en la siguiente dirección: Repositorio SolidarianID. Se trata de un repositorio autocontenido, es decir, contiene toda la configuración y los archivos necesarios para poder ejecutarse.

Dado que el interés de esta asignatura se centra en el *backend*, accedemos al correspondiente subdirectorio. Aquí, entre otros elementos, encontraremos un archivo README.md con

información relevante sobre dicho componente, así como las instrucciones necesarias para desplegarlo en modo producción.

Para ejecutarlo, es necesario contar al menos con **Docker**, **Docker Compose** y **Make** (opcional).

A continuación se indican los comandos que permiten levantar el sistema.

1. Crear la red necesaria para la comunicación de los servicios:
    - *make create-network*
    - Alternativa: *docker network create --driver bridge solidarianid-network*
2. Levantar los servicios en modo producción:
    - *make run-prod*
    - Alternativa: *docker-compose -f docker-compose-prod.yml up -d --build*
3. Vease el resto de comandos disponibles:
    - *make help*

La infraestructura de SolidarianID se organiza mediante un archivo *docker-compose-prod.yml*, que define los contenedores correspondientes a los servicios de mensajería, bases de datos y microservicios. El sistema utiliza Apache Kafka y Zookeeper para la gestión de mensajes asíncronos entre microservicios. Zookeeper se encarga de gestionar los nodos de Kafka (sólo poseemos uno) y está expuesto en el puerto 2181, mientras que Kafka opera como broker de mensajes en el puerto 9092.

En cuanto a las bases de datos, el microservicio de Usuarios utiliza PostgreSQL, una base de datos relacional expuesta en el puerto 5432, con almacenamiento persistente definido en el volumen postgres_data. El microservicio de Comunidades utiliza MongoDB, una base de datos NoSQL orientada a documentos, que opera en el puerto 27017 y almacena sus datos de manera persistente en mongo_data. El microservicio de Estadísticas utiliza Cassandra, una base de datos distribuida columnar expuesta en el puerto 9042, con persistencia de datos en cassandra_data. Debido a que, para Cassandra, no se utiliza ninguna librería que facilite la creación de las tablas en la base de datos a partir de los archivos de declaración de persistencia, ha sido necesario emplear la imagen *bitnami/cassandra*. Así, mediante la configuración de *volumes* podemos cargar automáticamente nuestro archivo *cassandra_schema.cql*, que define el esquema de la base de datos al momento de arrancar el contenedor. Los esquemas del resto de bases de datos son creados automáticamente al levantar la aplicación gracias a las librerías TypeORM y Mongoose.

El despliegue también incluye el API Gateway, que actúa como punto de entrada para los microservicios y se encuentra expuesto en el puerto 3000. Los microservicios están disponibles en los siguientes puertos: Users-MS (3001) para gestionar los usuarios y su historial de actividades, Communities-MS (3002) para la gestión de comunidades, causas y acciones, y Statistics-MS (3003) para el procesamiento y almacenamiento de estadísticas y reportes de la plataforma.

A continuación, en la Figura 5 se presenta un diagrama que representa la arquitectura dockerizada de SolidarianID y sus distintos componentes.

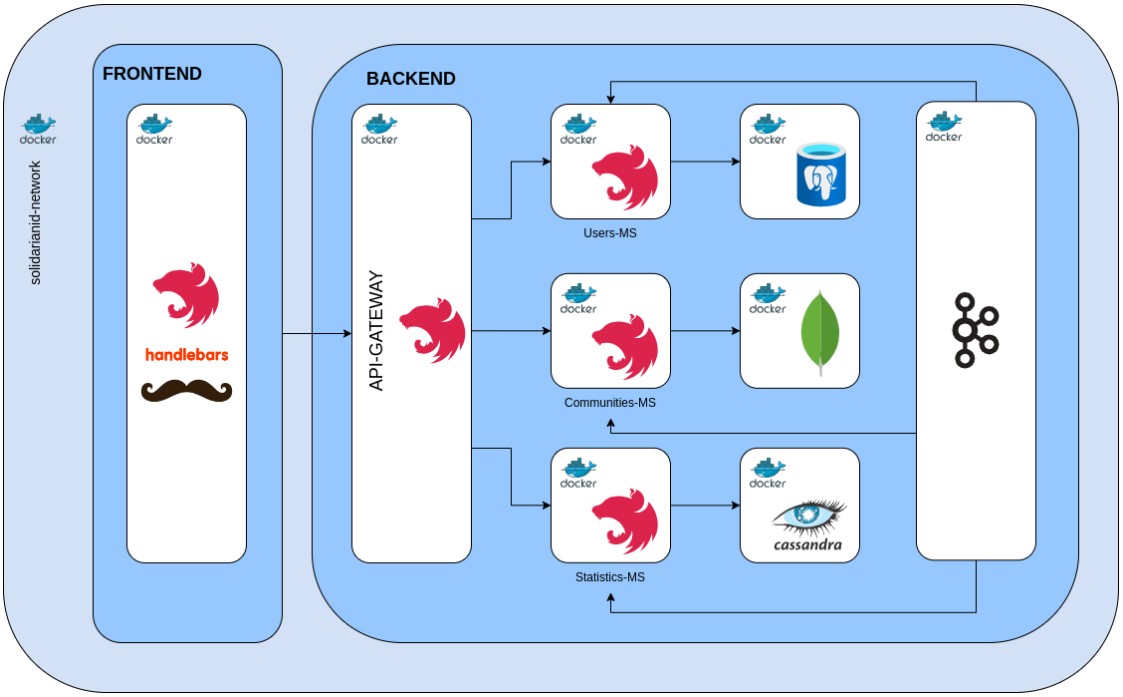

Figura 5. Arquitectura dockerizada de SolidarianID.

9 Conclusiones.

El desarrollo de esta práctica nos ha permitido implementar una arquitectura de datos distribuida basada en microservicios y comprender la importancia de seleccionar sistemas de almacenamiento según las necesidades de cada componente: PostgreSQL para garantizar consistencia en el microservicio de Usuarios, MongoDB por su flexibilidad en el manejo de entidades dinámicas en Comunidades, y Cassandra para gestionar grandes volúmenes de datos agregados en Estadísticas.

La integración de Kafka como broker de mensajería nos ha permitido asegurar una comunicación asíncrona eficiente y mantener la sincronización de los datos en tiempo real entre microservicios. Esta arquitectura distribuida nos brinda la capacidad de escalar cada microservicio de manera independiente según la carga de trabajo. Sin embargo, detectamos que la configuración actual de Kafka con un solo nodo representa un punto único de fallo, por lo que se debería aumentar el número de brokers en el clúster para mejorar la tolerancia a fallos y garantizar la alta disponibilidad del sistema.

10 Bibliografía.

1. Lima, T. (2022, enero 2). *NodeJS Microservices with NestJS, Kafka, and Kubernetes*. Lima's Cloud. https://limascloud.com/2022/01/02/nodejs-microservices-with-nestjs-and-kafka/ Lima's Cloud

2. Vural, H. (2023). *cassandra-for-nest*. GitHub. https://github.com/vural-hakan/cassandra-for-nest GitHub

3. MongoDB, Inc. (s.f.). *Documentación de MongoDB*. https://www.mongodb.com/docs/

4. PostgreSQL Global Development Group. (s.f.). *Documentación de PostgreSQL*. https://www.postgresql.org/docs/

5. Apache Software Foundation. (s.f.). *Documentación de Cassandra*. https://cassandra.apache.org/doc/latest/

6. NestJS. (s.f.). *TypeORM Integration*. https://docs.nestjs.com/techniques/database#typeorm-integration

7. NestJS. (s.f.). *MongoDB Integration*. https://docs.nestjs.com/techniques/mongodb

8. NestJS. (s.f.). *Microservices with Kafka*. https://docs.nestjs.com/microservices/kafka

9. Universidad de Murcia. (2024). *Recursos de la asignatura Arquitectura de Datos*. Máster en Ingeniería de Software (MISUM), curso 2024-2025.