https://drive.google.com/drive/folders/1RdmOTTZBnPP9axXRphZW-elzHYx23C1E

Punto 1 - Modelo de Datos (30 puntos)

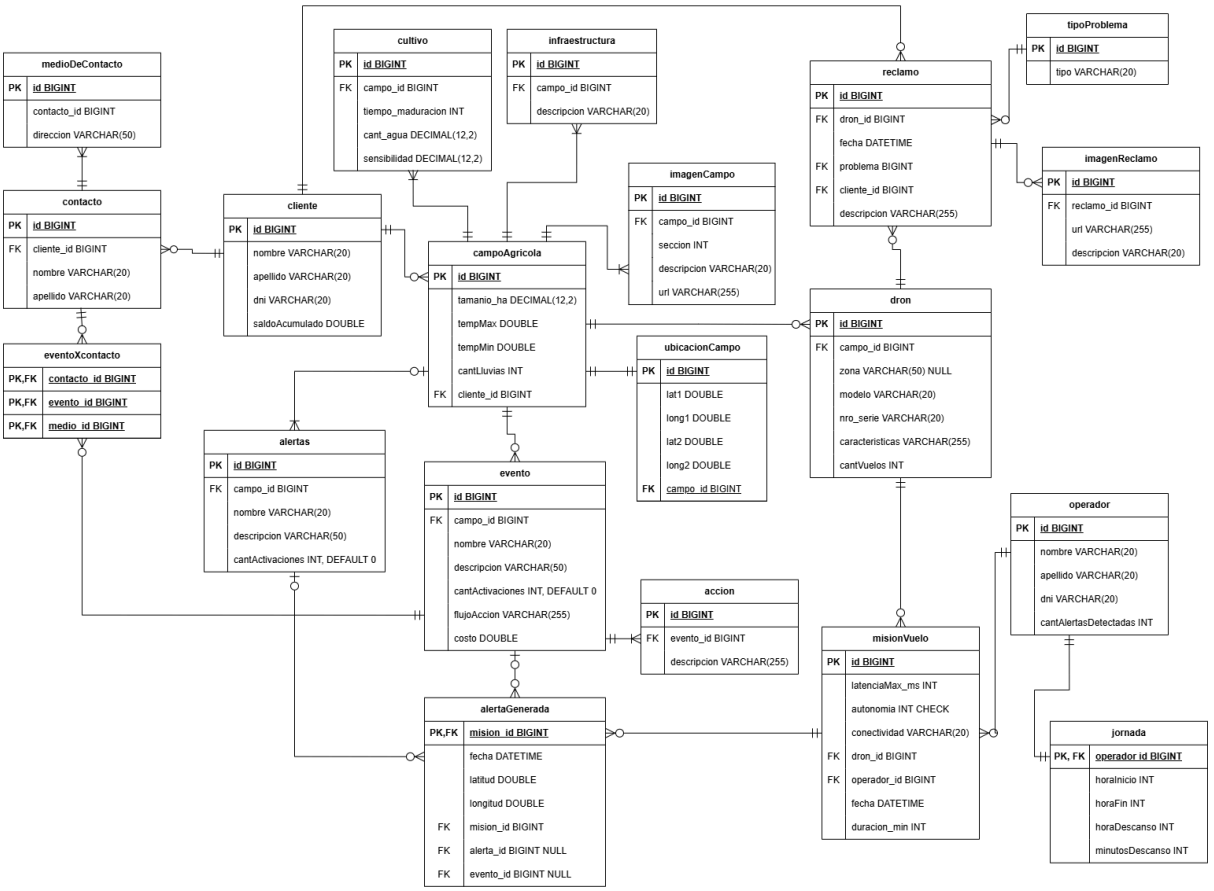
1) (10 puntos) Enumere las entidades identificadas y mencione, de forma concisa, qué representa cada una de ellas.

A gran escala, las principales entidades detectadas son:

- Cliente. Es quien se registra en la aplicación. Paga el servicio y carga su información personal, personalizan las alertas de sus campos.
    - Contacto: Los clientes poseen listado de personas a contactar ante alertas.
- Campos Agrícolas. Poseen una ubicación, infraestructura y cultivos. Los clientes cargan campos para ser monitoreados.
    - Imagen: Representa la url de la imagen cargada del campo.
    - Infraestructura: Los campos poseen diferentes infraestructuras con sus características.
    - Cultivo: Tipo de cultivo que se cosecha en el campo.
- Alertas. Las alertas ya están predefinidas en el sistema. Se activan cuando un suceso o dron las activa. Cada cliente tendrá sus alertas por campo.
- Eventos. Los eventos son personalizados y poseen un costo extra. Ejecutan un flujo de acciones tras un suceso disparador. Se puede notificar a la lista de contactos. Cada cliente tendrá sus eventos por campo.
- Dron. Son los dispositivos de monitoreo y detección. Realizan vuelos con frecuencia sobre los campos de los clientes. Son pilotados por supervisores.
- Operador/Supervisor. Son los empleados que manejan los drones y envían las alertas de ser necesario.
- Misión de vuelo: Posee un operador y dron con la cual se realiza. Las operaciones monitorean campos y generan alertas si encuentran algo.
    - AlertaGenerada: Se indica qué misión la generó, cuándo y dónde. También se indica si se trata de una Alerta o Evento. Cada AlertaGenerada dispara acciones diferentes.
- Reclamo. Los clientes pueden realizar reclamos ante fallas de los drones.
    - ImagenReclamo: Urls de fotos del problema. Son opcionales.
    - Tipo de Problema: Predefinidos en el sistema. Son categorías que permiten la mejor clasificación del reclamo (batería, conectividad, calibración, etc.).
- Administrador. Son quienes fijan precios, medios de contacto, definen alertas y atienden las necesidades del sistema.

2) (20 puntos) Realice el modelo de datos del dominio presentado. Indique todos los supuestos que crea necesario considerar. Además, resulta de suma importancia que justifique todas las decisiones de diseño que tomó.

https://drive.google.com/file/d/1kbIRo0WaV3y48SDJpJE6USL7dWud8yMy/view?usp=sharing



Decisiones del modelado:

Por primera forma normal, los campos multivaluados como Cultivos, Imágenes e Infraestructura serán normalizados en una tabla aparte. La ubicación se separa en 4 puntos cardinales para identificar las esquinas del campo.

Los clientes pueden registrar más de un campo, donde desean implementar el servicio y cargarle a ese campo diferentes alertas.

Los eventos son personalizados, tienen un costo y determinan un flujo de acciones a seguir cuando son disparados. Se puede configurar que los contactos reciban diferentes mensajes de eventos. Se realiza con una tabla intermedia eventoXcontacto.

Los contactos pueden tener múltiples medios de contacto, ya sean del tipo teléfono, email, o mensaje de texto. La tabla MedioDeContacto posee los diferentes medios que la persona tiene (pudiendo agregar más en el futuro).

Para trazabilidad, se indica que el dron registra los horarios de sus vuelos y qué operadores lo pilotaron. Para ello se hace la tabla MisionVuelo. Permite trazar también las condiciones cómo la conectividad, latencia, duración, etc.

Parece indicarse que el dron siempre está trabajando en un solo campo, pero que el campo puede tener más de un dron. Es una relación 1aN. Se agrega un campo opcional para indicar la zona dónde opera de ser necesario.

alertaGenerada registra todas las alertas y eventos que se produjeron, su ubicación exacta y fecha de ocurrencia.

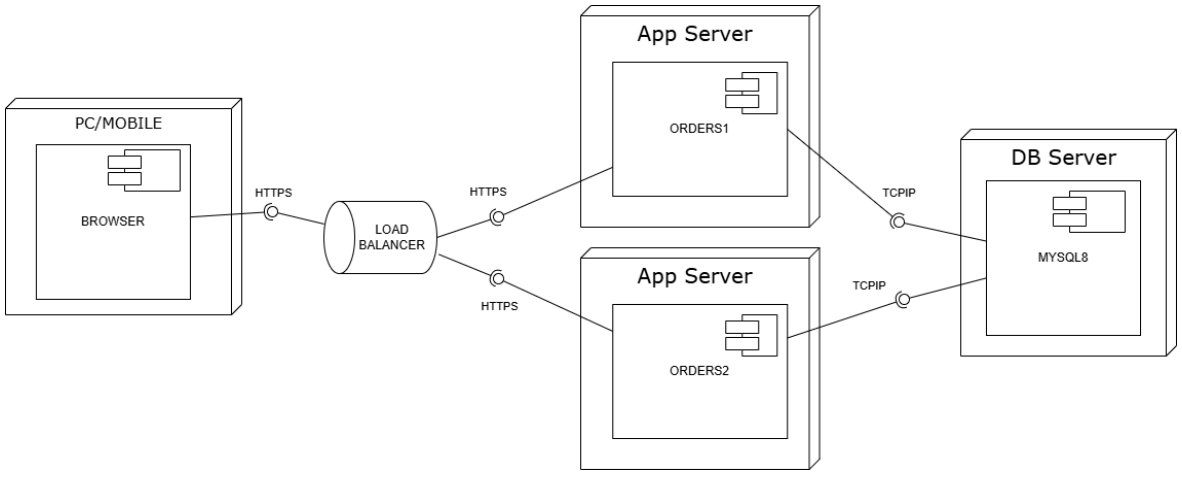
Para mejor performance y facilitar las estadísticas, se desnormalizaron datos como cantidad de vuelos o alertas resueltas.

Los supervisores pueden registrar sus horarios de trabajo, incluyendo inicios, finales y descansos en la tabla Jornada. Ej. inicio 8hs, fin 14hs. descanso a las 12hs, 30min.

Punto 2 - Arquitectura (40 puntos)

1) (10 puntos) Considerando el siguiente Diagrama de Componentes, donde se muestra la aplicación escalada en Microservicios, genere el **Diagrama de Despliegue** correspondiente pero considerando un Estilo Monolítico.

https://drive.google.com/file/d/1-IzX9WDy1xG9C-1rm6UbizE5vZ8oiSxw/view?usp=sharing



- Se incluyen los nodos sobre los cuales deployar.
- Se incluye un load balancer para distribuir cargas, ya que el diagrama de componentes poseía dos instancias de la app.
- Al ser un monolito, posee un único componente “silo” robusto.
- Se quitan los microservicios

2) (10 puntos) Considerando que al generarse una Alerta, otro Microservicio calcula el impacto a través de un algoritmo, para que el valor sea enviado al Cliente. El Microservicio de Alerta llama a través de una API REST al Microservicio de Cálculo.

a) ¿Cómo podría evitar que esta llamada encadenada genere un problema del Microservicio de Alerta?

Se puede solucionar con uno de los patrones de diseño orientados a microservicios: El **Circuit Breaker**. Funciona como un interruptor: si detecta muchos errores o demoras, “abre el circuito” y bloquea las llamadas por un tiempo, evitando sobrecargar el sistema y permitiendo que se recupere. Su objetivo es mejorar la resiliencia y evitar fallos en cascada entre microservicios.

De esta forma, el Microservicio de Alerta no se bloquea ni depende totalmente del de Cálculo, ya que puede:

- Devolver una respuesta temporal o degradada al cliente (“modo degradado”).
- Registrar la falla para ser gestionada posteriormente.
- Mantener su disponibilidad mientras el otro servicio se recupera.

Además, este patrón puede complementarse con otros mecanismos:

- Timeouts y reintentos controlados (Retry Pattern).
- Fallbacks, para retornar una respuesta alternativa.
- Bulkhead Pattern, para aislar recursos y evitar que una falla se propague.

b) ¿Es posible utilizar una Caché para mejorar esta situación?

No podrían almacenarse en Caché, ya que los parámetros de entrada son diferentes y se calculan en tiempo real durante cada alerta.

La caché es ideal para valores constantes calculados previamente. Reducen las consultas costosas repetitivas y la latencia de respuesta.

3) (10 puntos) Considerando que se generan Alertas que deben llegar a cada Cliente.
a) ¿Cómo haría este proceso?

Las misiones sobre los campos agrícolas generan eventos/alertas. Ese evento se publica de forma asíncrona en una cola de mensajes. Ventaja: desacopla productores y canales, tolera picos, permite retries, DLQ y trazabilidad.

1. Evento origen / **trigger** (operador agrícola detecta condición)
2. Se llama API POST /alertas al microservicio Alertas

3. **Alertas** valida reglas, persiste alerta en DB.
4. Crea *deliveryRequest* con destinatarios y canales
5. Publica mensaje en la cola: alertas.outbox o alertas.topic.

6. **Mensajería** consumer recibe mensaje:
7. Renderiza template (vía Template service o local),

8. Chequea rate limit / throttling
9. Intenta enviar por proveedor (SMTP, SMS gateway),
10. Registra resultado en tabla `deliveries` (SUCCESS / FAILED / RETRY),
11. Si falló transitorio -> reencola con backoff; si falló permanente -> DLQ y notifica.

b) Considerando una Arquitectura de Microservicios ¿Qué microservicios involucraría?

Microservicios involucrados:

- Un microservicio Alertas se encarga de validar/normalizar y publicar el evento. Recibe el trigger, valida reglas de negocio, persiste la alerta generada y publica el evento en la cola.

- Un microservicio Mensajería toma los mensajes de la cola y envía por email / SMS usando proveedores externos (SMTP, Twilio, etc.).

- Gateway / API Gateway: Punto de entrada de clientes externos para crear alertas (los operadores en sus misiones).

- Auth / IAM: Microservicio de autenticación/autorización y emisión de tokens.

c) ¿Qué mecanismo de integración utilizará entre los servicios?

Mensajería asíncrona basada en colas:

- RabbitMQ: fácil de usar, patterns (work queues, routing, TTL, DLQ), buen para delivery at-least-once con ack. Ideal para casos con varios consumidores heterogéneos y control de reintentos/orden.

- Kafka: throughput alto, retención, ideal cuando necesitas reproc/analytics y ordering por key. Menos natural para retry/dlq (requiere patterns).

- Cloud queues (SQS+SNS): gestión administrada, escalado simple.

Formato de mensajes: JSON con un schema versionado (o Protobuf si se usa gRPC/Kafka).

Integración sincrónica mínima: Alertas puede exponer REST (o gRPC) para crear alertas; la comunicación interna usa la cola.

4) (10 puntos) Los videos generados por los diferentes Drones son almacenados en los servidores de aplicación. Se detectó que la reproducción de los mismos desde diferentes ubicaciones de forma simultánea genera problemas de performance de la aplicación general.

a) ¿Qué propondría para resolverlo?

Propondría utilizar una CDN (Content Delivery Network) para distribuir los videos de forma eficiente y evitar que todas las solicitudes lleguen directamente al servidor de aplicación.

Una CDN es una red de servidores distribuidos geográficamente que almacenan copias (caché) del contenido estático (en este caso, los videos generados por los drones) y los sirven al usuario desde el nodo más cercano.

Ventajas:

- Reduce la carga del servidor de aplicación, que deja de manejar tráfico pesado de video.
- Mejora la latencia y velocidad de reproducción, ya que los usuarios descargan desde un nodo cercano.
- Escala automáticamente ante múltiples reproducciones simultáneas sin saturar la infraestructura.
- Aumenta la disponibilidad, ya que si un nodo falla, otro puede servir el contenido.

En resumen, el servidor de aplicación sólo se ocupa de generar y subir el video a un repositorio central (o bucket), y la CDN se encarga de distribuirlo y servirlo a los usuarios finales.

b) ¿Qué cambios debería realizar sobre la Arquitectura del Sistema Actual?

Para incorporar una CDN, los principales cambios serían los siguientes:

1. Separar el contenido estático (videos) del backend de la aplicación:

Actualmente, los videos se almacenan directamente en los servidores de aplicación. Se deben mover a un repositorio de archivos centralizado, que puede ser local o externo en la nube (ej. Amazon S3, Google Cloud Storage o Azure Blob Storage).

2. Integrar la CDN

Configurar la CDN (por ejemplo, CloudFront, Cloudflare, Akamai, etc.) para que distribuya el contenido desde ese repositorio. Definir políticas de cacheo, expiración y actualización (TTL).

3. Actualizar la arquitectura lógica

El servidor de aplicación se enfocará sólo en manejar la lógica del sistema (autenticación, gestión de drones, metadatos de los videos, etc.), ya no el streaming. La CDN se conecta directamente con los clientes para entregar los archivos multimedia.

Punto 3 - Persistencia (20 puntos)

```
@Entity @Table(name="persona")
class Persona {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @Column(name = "denominacion")
  private String denominacion;

  @Enumerated (EnumType.STRING)
  @Column(name="tipo",nullable = false)
  private TipoPersona tipo;
}

@Entity @Table(name="proyecto")
class Proyecto {
  @Id @GeneratedValue(strategy = GenerationType.IDENTITY)
  private Integer id;

  @Column(name = "fechaComienzo")
  private LocalDate fechaComienzo;

  @Column(name = "nombre")
  private String nombre;

  @Column(name = "descripcion")
  private String descripcion;

 @OneToMany(mappedBy = "persona")
 private List<Persona> interesados;

 @OneToMany(mappedBy = "persona")
 private List<Persona> aportantes;

 @OneToMany(mappedBy = "persona")
 private List<Persona> promotores;

 @Column(name = "porcentajeDeAvance")
 private Double porcentajeDeAvance;

 @ManyToOne
 @JoinColumn(name = "criterio_id")
 private CriterioCalculoAvance criterioCalculoAvance;

 @OneToMany(mappedBy = "hito")
 private List<Hito> hitos;
}
```

```java
@Entity @Table(name="hito")
class Hito {
  @Id @GeneratedValue
  private Integer id;

  @Column(name = "descripcion")
  private String descripcion;

  @Column(name = "fechaHora")
  private LocalDateTime fechaHora;

  @ManyToMany
  private List<Hito> hitosDependientes;

  @Enumerated (EnumType.STRING)
  @Column(name="estadoActual",nullable = false)
  private EstadoHito estadoActual;
 }

@MappedSuperclass
public abstract class CriterioCalculoAvance {
  @Id  @GeneratedValue
  private Long id;
}

@Entity  @Table(name="cantAportantesProyecto")
 public class CantAportantesProyecto extends CriterioCalculoAvance {
  @Column(name="cantidadMaxima")
  private Integer cantidadMaxima;
}

@Entity  @Table(name="cantHitosAlcanzados")
 public class CantHitosAlcanzados extends CriterioCalculoAvance {
   @Column(name="cantidadMinima")
   private Integer cantidadMinima;
}

@Entity  @Table(name="hitosEspecificosAlcanzados")
 public class HitosEspecificosAlcanzados extends CriterioCalculoAvance {
   @OneToMany(mappedBy = "hito")
   private List<Hito> hitosAlcanzados;
}
```

Punto 4 - Teoría (10 puntos)

1) (5 puntos) V o F. Justifique su respuesta.
a) La utilización de frameworks de ORM mejora notablemente la performance de las consultas a una BD.

Falso. Los ORM suelen reducir levemente la performance porque agregan una capa de abstracción entre el código y la base de datos. Sin embargo, mejoran la productividad, la mantenibilidad y la integridad de los datos, evitando errores de SQL manual y gestionando automáticamente las relaciones.

b) Si recibo un código de estado del protocolo HTTP 403, me está indicando que no puedo acceder al recurso por falta de credenciales en la solicitud.

Falso. El código 403 (Forbidden) indica que el servidor entendió la solicitud, pero el cliente no tiene permisos para acceder al recurso, aun cuando esté autenticado.

La falta de credenciales se indica con 401 (Unauthorized).

2) (5 puntos) Ejemplifique:
a) Dos componentes arquitectónicos utilizados en el mercado

API Gateway: gestiona el ingreso de peticiones a múltiples microservicios.

Message Broker: componente que permite comunicación asincrónica entre servicios (ej: RabbitMQ, Kafka).

b) Dos ORM de mercado

Hibernate (Java)
Entity Framework (C# / .NET)
Mongoose (Node.js / JavaScript)

c) Una forma de deploy de microservicios

Contenedores con Docker y orquestación con Kubernetes, donde cada microservicio se ejecuta en su propio contenedor y se gestiona de forma independiente.