TicketGod

Tomas Alvarado, Pablo de Leon

Diseño y Programación O.O

Universidad de Los Andes

Camilo Ortiz Casas

2025

Contexto

El proyecto implementa una plataforma de boletería que gobierna, de forma trazable y auditable, todo el ciclo de un evento: creación por parte del organizador, definición de venues y localidades, publicación de la oferta, emisión y venta de tiquetes, transferencia de titularidad, cancelaciones y reembolsos, y elaboración de reportes financieros. La solución se apoya en un modelo de dominio en Java que estructura entidades y reglas de negocio de manera explícita: un Evento se realiza en un Venue aprobado, en una fecha y franja exclusivas para evitar solapamientos; su oferta se descompone en Localidades que pueden ser numeradas, donde existe un mapa de asientos que garantiza la unicidad de asignación, o no numeradas, donde el control se ejerce por cupos disponibles. Los tiquetes poseen un identificador único verificable, un precio final calculado a partir de precio base más cargo porcentual por servicio y cuota fija de emisión, y estados coherentes con su ciclo de vida (emitido, transferido, reembolsado, anulado). La solución contempla variantes funcionales de tiquete: el Básico, que en localidades numeradas porta un número de asiento; el Múltiple, que agrupa entradas que se venden y usan como una unidad lógica (pases o palcos), con reglas y topes a nivel de paquete; y el Deluxe, que añade beneficios y prohíbe la transferencia parcial por control de valor y seguridad.

El ecosistema define tres roles con límites nítidos. El cliente compra, transfiere y consulta su historial; el organizador crea eventos, parametriza localidades y precios, y consulta resultados; y el administrador establece políticas económicas (porcentajes de servicio y cuota fija), aprueba venues, autoriza cancelaciones y consolida las finanzas de la tiquetera. Dos decisiones de diseño acotan responsabilidades y simplifican gobierno: el organizador no puede comprar desde su propio perfil tiquetes de eventos ajenos (si desea hacerlo, debe operar como cliente con credenciales separadas), y existirá un único usuario Administrador compartido, con la misma credencial y el mismo alcance funcional. La experiencia de pago se delega a una pasarela externa; la plataforma no procesa medios de pago, pero sí garantiza idempotencia e integridad en la emisión, de modo que una orden, aun con reintentos o latencias, nunca produzca duplicidades. La persistencia se implementa mediante base de datos embebida (SQLite), lo que permite validar transacciones reales, restricciones de unicidad (ID de tiquete y par localidad–asiento) y consistencia de estados. Para acelerar pruebas y refinamiento de reglas, el proyecto incorpora una consola de línea de

Alcance

El alcance describe lo que hará la plataforma en esta versión, cómo lo hará y hasta dónde llega. En primer término, se habilita el gobierno de la oferta: registro y aprobación de venues con su ubicación, capacidad y restricciones, y creación de eventos con exclusividad temporal por venue. Cada evento define sus localidades; cuando son numeradas, el sistema mantiene un mapa de asientos y valida la unicidad en el momento de vender; cuando son no numeradas, controla capacidad para impedir sobreventa. Sobre esa estructura, el organizador asocia tipos de tiquete con sus reglas particulares. El cálculo del precio final queda parametrizado por el administrador a través de políticas globales de cargo porcentual por servicio y cuota fija de emisión, y el resultado se persiste junto con el identificador único de cada tiquete; esta trazabilidad asegura explicabilidad de cobros y auditoría contable.

En segundo término, se soporta la emisión y venta integradas a la pasarela externa. Cada intento de compra se identifica con una clave de idempotencia y se ejecuta dentro de una transacción que valida simultáneamente capacidad/asientos y estados de inventario. Si todas las precondiciones se cumplen, se generan los tiquetes y se registran en la base; si alguna condición falla, la operación se revierte sin efectos colaterales y se devuelve un mensaje claro. Este mismo rigor aplica a la transferencia de titularidad: el emisor se autentica, el sistema verifica la elegibilidad del tipo de tiquete (por ejemplo, bloquea traslados parciales de *Deluxe*), registra el traspaso con sello de tiempo y, si corresponde, solicita aceptación del receptor. El perfil de organizador está explícitamente impedido de comprar eventos ajenos desde su rol, con lo cual se evita mezclar responsabilidades y se protege la coherencia del flujo de compra.

En tercer término, se habilitan cancelaciones y reembolsos conforme a política. El administrador puede cancelar un evento por causales justificadas o aprobar solicitudes del organizador; la plataforma actualiza el estado de tiquetes afectados, acredita reembolsos al saldo virtual del cliente o los gestiona vía pasarela, y conserva la trazabilidad contable para conciliación y reporte. De manera transversal, la solución produce reportes financieros para organizadores (ventas por evento, localidad y tipo de tiquete) y para la administración (ingresos por servicio y emisión), respetando la separación contable entre los ingresos del organizador y los de la tiquetera. La operación se protege mediante controles de acceso por rol y bitácoras de auditoría; en datos, se aplican índices y restricciones que preservan integridad, y en lógica de aplicación se concentran las reglas de negocio en servicios que evitan desvíos (precio, compra por rol, transferencia, cancelación y reembolso).

Finalmente, se establecen límites para enfocar el desarrollo en el núcleo transaccional: quedan fuera de alcance el control en puerta (escaneo QR/NFC, torniquetes, validación offline), la impresión material de tiquetes por parte de la plataforma, el mercado secundario o reventa oficial, la tarificación dinámica por demanda y la administración multiusuario; en su lugar se adopta un único administrador con credenciales compartidas. Estas exclusiones reducen complejidad técnica, operativa y regulatoria, permiten canalizar la lógica ya codificada hacia un producto completo y estable, y dejan puntos de extensión definidos para evolucionar la solución sin reescrituras profundas.

Objetivos

La plataforma debe garantizar la emisión correcta y única de tiquetes bajo cualquier condición de concurrencia. Para ello, la confirmación de una orden se realizará de forma atómica e idempotente: una misma clave de orden no puede producir más de un tiquete, y los intentos repetidos nunca dejarán estados inconsistentes. En localidades numeradas, todo asiento tendrá asignación exclusiva; en no numeradas, el sistema impedirá la sobreventa aplicando validaciones de capacidad al momento de la transacción. El precio final de cada tiquete será calculado y persistido con exactitud a partir de sus componentes (base, servicio y emisión) y quedará disponible para auditoría y reporte financiero sin ambigüedades.

El límite entre roles será verificable en tiempo de ejecución: el organizador sólo operará sobre su propia oferta y, cuando desee comprar eventos ajenos, deberá autenticarse como cliente. La transferencia de titularidad preservará integridad y trazabilidad (quién, qué y cuándo), aplicando las restricciones del tipo de tiquete —incluida la prohibición de transferencia parcial en Deluxe— y actualizando el estado de origen/destino sin posibilidad de duplicación. Ante cancelaciones, el sistema ejecutará reembolsos conforme a política, bloqueará los tiquetes afectados y registrará movimientos de forma que los ingresos por servicio de la tiquetera queden separados de las ventas brutas del organizador.

Desde lo operativo, la solución mantendrá seguridad básica (cifrado en tránsito y reposo, acceso por rol), registro exhaustivo de operaciones relevantes (compras, transferencias, reembolsos) y reportes por evento, localidad y organizador. A nivel de ingeniería, contará con migraciones de esquema o scripts reproducibles, pruebas sobre los flujos críticos (venta numerada/no numerada, idempotencia, transferencia con restricciones y cancelación con reembolso) y una interfaz de consola que permita validar el comportamiento de punta a punta mientras se completa la aplicación.

No objetivos

En esta fase operaremos con un único usuario Administrador. Centralizar el gobierno en una sola credencial reduce la complejidad de configuración y la superficie de error mientras consolidamos el núcleo transaccional Evitamos así matricular roles y permisos heterogéneos, auditorías por usuario y flujos de recuperación de credenciales antes de tiempo. El costo de oportunidad es renunciar, temporalmente, a la segregación granular de funciones

De igual forma, el organizador no compra eventos de terceros desde su propio perfil. Exigirle que, si desea comprar ajeno, inicie sesión como cliente separa nítidamente la oferta que administra del consumo que realiza, elimina conflictos de interés y facilita las validaciones y auditorías. Con esta decisión reducimos riesgos de abuso (por ejemplo, saltarse topes o acceder a inventario privilegiado) y mantenemos simples los invariantes de negocio.

También queda fuera de alcance el control en puerta (escaneo QR/NFC, torniquetes y operación offline). Implementar acceso físico exige hardware, procedimientos y tolerancia a fallas de conectividad que no aportan ventaja mientras el backoffice aún se está consolidando. Al posponerlo, evitamos incidentes de ingreso, sincronizaciones complejas y costos operativos; dejaremos, no obstante, el estado del tiquete y su identificador listos para que un verificador futuro consulte en tiempo real o con listas firmadas.

No abordaremos la impresión física de tiquetes. Limitar la emisión del comprobante digital verificable elimina costos y logística de impresión, cadena de custodia y reclamaciones por deterioro. Si un organizador lo exige en etapas posteriores, se podrá integrar un proveedor especializado o habilitar exportables con controles antifraude, sin reescribir el núcleo.

El mercado secundario o reventa oficial tampoco se incluye. Diseñarlo bien implica reglas de autenticidad, límites para evitar *scalping*, reemisiones y comisiones, además de tensiones regulatorias con los organizadores.

Diagramas de contexto del sistema

Información sobre la persistencia

La plataforma utiliza una base de datos relacional embebida (SQLite) para respaldar todos los flujos críticos —venta, transferencia, cancelación y reembolso— con propiedades ACID y trazabilidad completa. El esquema se organiza en torno a los agregados del dominio y mantiene integridad referencial, restricciones de unicidad y transacciones que garantizan resultados atómicos: o bien una operación se confirma por completo, o no deja efectos parciales.

El núcleo lo conforman tablas para eventos, venues y localidades. En localidades numeradas se gestiona un mapa de asientos en asientos (una fila por asiento con su estado), mientras que en no numeradas la propia localidades mantiene capacidad, vendidos y disponible. La tabla tiquetes almacena la identidad única del tiquete, su estado de ciclo de vida (emitido, transferido, reembolsado, anulado), su vínculo a evento y localidad, y el precio final junto a los componentes de cálculo (precio base, porcentaje de servicio y cuota fija de emisión) para asegurar explicabilidad. Las órdenes de compra se registran en ordenes con su clave de idempotencia y estados de autorización; orden_items detalla los tiquetes pedidos. Las transferencias y reembolsos se modelan como movimientos con sello de tiempo y referencias a los actores y al motivo, de modo que la auditoría pueda reconstruir cualquier operación. Para políticas económicas y parámetros globales se emplea config_admin.

Cada tiquete tiene un id_tiquete único; en numeradas, el par (id_localidad, nro_asiento) es único para impedir colisiones de asignación; en no numeradas, las operaciones de venta actualizan disponible de forma condicionada para evitar sobreventa. La integridad entre eventos, venues, localidades, asientos y tiquetes se preserva con claves foráneas y reglas de borrado/actualización restringidas (no se eliminan registros con dependencias activas; se recurre a cambios de estado antes que a borrados físicos). Estados y tipos (p. ej., clase de tiquete) se limitan por checks o catálogos (lookup tables) para evitar valores inválidos.

Las rutas críticas (venta, transferencia, cancelación, reembolso) se ejecutan dentro de transacciones explícitas. En la venta: se verifica elegibilidad (rol, reglas del tipo de tiquete), se valida capacidad o asiento libre, se calcula y persiste el precio final, y se emite el tiquete; cualquier fallo revierte todo el bloque. La idempotencia se garantiza registrando en ordenes una clave única por operación: si la pasarela reintenta o el cliente repite la solicitud, el sistema devuelve el resultado ya emitido y jamás duplica tiquetes. En transferencias, la transacción mueve la titularidad, registra la traza y actualiza estados; en cancelaciones y reembolsos, cambia el estado de los tiquetes afectados, registra el movimiento contable y deja la operación en un punto consistente incluso ante fallas intermedias.

Además de las claves primarias, se indexan campos de alta selectividad y acceso frecuente: id_evento y id_localidad en tiquetes, (id_localidad, nro_asiento) en asientos, clave_idempotencia en ordenes, y campos de búsqueda para reportes (fecha de evento,

organizador, tipo de tiquete). Estos índices equilibran lectura de reportería y escritura transaccional, manteniendo tiempos consistentes en picos de demanda.

Todos los datos viajan bajo cifrado en tránsito y se almacenan con cifrado en reposo cuando el entorno lo permite. El control de acceso por rol limita operaciones a Cliente, Organizador y Administrador según sus capacidades; la bitácora de auditoría conserva marcas de quién, cuándo y qué cambió en compras, transferencias, cancelaciones y reembolsos. En términos de retención, los datos operativos necesarios para conciliación y reportes se conservan por el plazo definido por la operación; los registros técnicos de depuración y telemetría mantienen una ventana más corta y rotan automáticamente.

La base se acompaña de scripts de migración (DDL) reproducibles que permiten crear y versionar el esquema en ambientes de desarrollo y producción. Las modificaciones estructurales (nuevos tipos de tiquete, atributos o estados) se realizan mediante migraciones incrementales, cuidando la compatibilidad hacia atrás y, cuando sea necesario, rutinas de backfill para rellenar datos derivados (por ejemplo, materializar el precio final histórico). Este enfoque evita “drifts” entre entornos y facilita pruebas automatizadas sobre la persistencia.

Código relevante

Compra de tiquete en UsuarioComprador

```java
public List<Tiquete> comprarTiquete(Localidades localidad, int cantidad, double porcentajeServicio, double cobroEmision ) throws CapacidadExcedidaLocalidad{
    double precioBaseUnitario = localidad.getPrecioFinal();
    double costoServicioUnitario = precioBaseUnitario * porcentajeServicio;
    double precioFinalunitario = precioBaseUnitario + costoServicioUnitario + cobroEmision;

    double costoTotalTransaccion = precioFinalunitario * cantidad;

    Evento evento = localidad.getEvento();

    if (!localidad.verificarDisponibilidad(cantidad)) {
         throw new CapacidadExcedidaLocalidad(
            "No hay disponibilidad para " + cantidad +
            " tiquetes en esta localidad:  " + localidad.getNombreLocalidad()
         + " Crack.! :(");
    }

    this.saldo = saldo - costoTotalTransaccion;

    List<String> asientosAsignados = localidad.venderTiquetes(cantidad); // si es NUmerada tiene uun asiento si no es una lista de NULL.

    // ahora si crear los eventos uff

    List<Tiquete> tiquetesCreados = new ArrayList<>();

    for (String asiento: asientosAsignados) {

        Tiquete nuevoTiquete = new Basico(precioBaseUnitario,porcentajeServicio, cobroEmision, evento.getFecha(), this, localidad, evento, "ACTIVO", asiento);
        this.tiquetesComprados.add(nuevoTiquete);
        tiquetesCreados.add(nuevoTiquete);


    }


    return tiquetesCreados;

}
```

Transferir Tiquete en UsuarioComprador

```java
@Override
public void transferirTiquete(Tiquete tiquete, String passwordConfirmacion, String loginDestinatario,List<Usuario> todosLosUsuarios) throws AutenticacionFallidaException, TiqueteNoTransf

    if (!this.tiquetesComprados.contains(tiquete)) {
        throw new TiqueteNoTransferibleException("El tiquete " + tiquete.getIdTiquete() + " no te pertenece.");
    }

    if (!this.contrasena.equals(passwordConfirmacion)) {
        throw new AutenticacionFallidaException("Contraseña incorrecta. Transferencia cancelada.");
    }

    Usuario destinatario = null;
    for (Usuario u : todosLosUsuarios) {
        if (u.getLogIn().equals(loginDestinatario)) {
            destinatario = u;
            break;
        }
    }

    if (destinatario == null) {
        throw new Exception("Usuario destinatario '" + loginDestinatario + "' no encontrado.");
    }

    if (destinatario instanceof Administrador) {
        throw new TiqueteNoTransferibleException("No se pueden transferir tiquetes a un Administrador.");
    }

    tiquete.transferirTiquete(destinatario);
    this.tiquetesComprados.remove(tiquete);

    if (destinatario instanceof UsuarioComprador) {

        ((UsuarioComprador) destinatario).getTiquetesComprados().add(tiquete);

    } else if (destinatario instanceof OrganizadorEventos) {

        ((OrganizadorEventos) destinatario).getTiquetesComprados().add(tiquete);
    }


}
```

Cancelar Evento en Administrador

```java
public void cancelarEvento(Evento evento, List<Tiquete> todosLosTiquetesVendidos, boolean isCancelacionPorAdmin)  {

    evento.setEstado("CANCELADO");

    for (Tiquete tiquete :todosLosTiquetesVendidos ) {

        if (tiquete.getEvento() != null && tiquete.getEvento().equals(evento)) {

            String estadoActual = tiquete.getEstado();

            if (estadoActual.equals("ACTIVO") || estadoActual.equals("TRANSFERIDO")) {

                double montoReembolso = 0.0;

                if (isCancelacionPorAdmin) {
                    // Regla Admin: precio pagado (base + servicio) menos emisión
                    montoReembolso = tiquete.getPrecioBase() + tiquete.getCostoServicio();

                } else {
                    // Regla Organizador: solo precio base
                    montoReembolso = tiquete.getPrecioBase();
                }


                if (montoReembolso > 0) {
                    Usuario dueno = tiquete.getCliente();
                    dueno.setSaldo(dueno.getSaldo() + montoReembolso);
                }

                tiquete.setEstado("REEMBOLSADO");

            }
        }
    }

}
```

Calcular Ganancias en Administrador

```java
public Map<String, Double> calcularGanancias( List<Tiquete> todosLosTiquetesVendidos, List<Evento> todosLosEventos) {

        Map<String, Double> reporte = new HashMap<>();
        double gananciasTotales = 0.0;

        // Mapa para guardar ganancias por organizador
        Map<OrganizadorEventos, Double> gananciasPorOrganizador = new HashMap<>();

        // Iteramos sobre TODOS los tiquetes vendidos
        for (Tiquete tiquete : todosLosTiquetesVendidos) {

            String estado = tiquete.getEstado();

            if (!estado.equals("CORTESIA") && !estado.equals("REEMBOLSADO")) {

                double gananciaEsteTiquete = tiquete.getCostoServicio() + tiquete.getCostoEmision();

                //Sumar a las ganancias totales
                gananciasTotales += gananciaEsteTiquete;

                //Sumar a las ganancias por evento
                Evento evento = tiquete.getEvento();
                if (evento != null) {
                    String claveEvento = "GANANCIA_EVT_" + evento.getNombre();
                    double gananciaEventoActual = reporte.getOrDefault(claveEvento, 0.0);
                    reporte.put(claveEvento, gananciaEventoActual + gananciaEsteTiquete);

                    //Sumar a las ganancias por organizador
                    OrganizadorEventos promotor = evento.getPromotor();
                    double gananciaPromotorActual = gananciasPorOrganizador.getOrDefault(promotor, 0.0);
                    gananciasPorOrganizador.put(promotor, gananciaPromotorActual + gananciaEsteTiquete);
                }
            }
        }

        // Guardar las ganancias totales
        reporte.put("GANANCIA_TOTAL_TIQUETERA", gananciasTotales);

        // Guardar las ganancias por organizador en el reporte principal
        for (Map.Entry<OrganizadorEventos, Double> entry : gananciasPorOrganizador.entrySet()) {
            String clavePromotor = "GANANCIA_PROMOTOR_" + entry.getKey().getLogIn();
            reporte.put(clavePromotor, entry.getValue());
        }
        return reporte;
    }
```

Guardar Venue en VenueDAO

```java
public Venue guardarVenue(Venue venue) throws SQLException {
    // El SQL para insertar en la tabla Venue.
    // No pasamos 'id_venue' porque es AUTOINCREMENT.
    String sql = "INSERT INTO Venue (tipo, ubicacion, capacidad_maxima, estado) VALUES (?, ?, ?, ?)";

    Connection conn = null;
    PreparedStatement pstmt = null;

    try {
        conn = ConexionSQLite.conectar();

        // Pedimos que nos devuelva las claves generadas (el id_venue)
        pstmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS);

        // Asignamos los valores a los '?'
        pstmt.setString(1, venue.getTipo());
        pstmt.setString(2, venue.getUbicacion());
        pstmt.setInt(3, venue.getCapacidadMaxima());
        pstmt.setString(4, venue.getEstado()); // "PENDIENTE" o "APROBADO"

        int affectedRows = pstmt.executeUpdate();

        // Verificamos que se haya insertado
        if (affectedRows == 0) {
            throw new SQLException("Error al guardar el venue, no se insertaron filas.");
        }

        // Recuperamos el ID autogenerado
        try (ResultSet generatedKeys = pstmt.getGeneratedKeys()) {
            if (generatedKeys.next()) {
                int idGenerado = generatedKeys.getInt(1);
                venue.setIdVenue(idGenerado);
                // Aquí podríamos actualizar el objeto 'venue' con el ID de la BD si fuera necesario,
                // pero nuestro modelo de Java no usa el ID numérico, así que solo confirmamos.
                System.out.println("Venue guardado con ID de BD: " + generatedKeys.getInt(1));
            } else {
                throw new SQLException("Error al guardar el venue, no se obtuvo el ID.");
            }
        }

        return venue; // Devolvemos el objeto original
```

Alternativas consideradas

Se evaluó representar todos los tiquetes con una sola clase parametrizada por “flags” (múltiple, deluxe, numerado). La ventaja era reducir el número de tipos y acelerar el modelado inicial; sin embargo, introducía condicionales frágiles en operaciones críticas (transferencia, reembolso, validación de parcialidad) y aumentaba el riesgo de violar reglas específicas. Se adoptó el enfoque de especializaciones (Básico, Múltiple y Deluxe), que expresa de forma explícita las invariantes y simplifica las validaciones en tiempo de ejecución.

Gobernanza de roles (multi-admin desde el inicio vs. administrador único). Un esquema completo de RBAC/ABAC con jerarquías permitiría segregación de funciones, pero aumentaba la complejidad de permisos, auditorías por usuario y soporte. Para el MVP se adoptó administrador único (una credencial compartida).

Compra desde perfil de organizador (permitir mixto vs. separar roles). Permitir que el organizador compre eventos de terceros desde su mismo perfil simplificaba la UX, pero generaba conflictos de interés y rutas de abuso (acceso a inventario privilegiado, bypass de topes). Se exigió la compra como cliente con credenciales separadas para preservar límites y auditoría.

Persistencia (JSON por archivos vs. SQLite embebido). Se evaluó almacenar la información en archivos JSON (uno por agregado o colección) por su baja fricción inicial, legibilidad humana y facilidad para semillas de datos en desarrollo. Esta opción elimina la necesidad de un motor relacional y reduce la curva de entrada, pero carece de propiedades ACID, no ofrece bloqueos ni transacciones reales, y complica garantías críticas del dominio: idempotencia fuerte por orden, unicidad (ID de tiquete y par localidad–asiento), consistencia bajo concurrencia y reversiones atómicas ante fallos. Además, las consultas para reportería (ventas por evento/localidad/tipo, conciliación) se vuelven costosas al no existir índices ni joins, y aumenta el riesgo de corrupción en escrituras parciales o accesos simultáneos. Por estas razones se eligió SQLite embebido, que aporta transacciones, restricciones de unicidad, integridad referencial e índices con un costo operativo muy bajo y excelente reproducibilidad local.

Preocupaciones transversales

La seguridad de la información es una preocupación central: el sistema maneja órdenes, estados de tiquete y parámetros económicos, por lo que debe evitar exposición de credenciales, fugas en tránsito o en reposo y filtración accidental en registros. Esto implica atender la confidencialidad de los datos de operación, la integridad de lo emitido y la autenticidad de quien ejecuta acciones sensibles, sin asumir confianza implícita entre componentes.

La transaccionalidad de los flujos críticos es innegociable. Venta, transferencia, cancelación y reembolso no pueden dejar estados intermedios ni efectos parciales. La preocupación es garantizar atomicidad y aislamiento incluso con concurrencia, latencias de la pasarela o reintentos de cliente, de manera que el resultado sea binario y estable: o se emite, o se revierte sin residuos.

La idempotencia es un riesgo recurrente en integración con pasarelas. Es una preocupación que la misma orden no genere múltiples emisiones si se repite la confirmación o se reenvía un webhook. La plataforma debe poder reconocer operaciones previamente resueltas y responder de forma determinística para prevenir duplicados y divergencias contables.

La integridad referencial y del dominio requiere atención continua. Es una preocupación que no existan asignaciones duplicadas de asiento en localidades numeradas, que no ocurra sobreventa en no numeradas y que los estados de tiquete sean mutuamente excluyentes y consistentes con su historial. Cualquier relajación aquí degrada confianza operativa y dificulta auditorías.

La gobernanza de acceso y separación de roles requiere límites claros. Es una preocupación que el organizador no pueda operar fuera de su ámbito, que el administrador no sea un punto único de fallo por falta de controles y que el cliente no eluda restricciones del tipo de tiquete. Cualquier ambigüedad en esta frontera abre la puerta a abuso o a errores operativos.

La experiencia de error incide directamente en operación. Mensajes vagos o poco accionables llevan a reintentos peligrosos y a estados duplicados. Es una preocupación que cada fallo sea explícito en su causa y sugiera la acción correcta para no escalar el problema a niveles de datos.