Evaluación Práctica – Agrotech

Nombre: Fernando Camacho **Fecha:** 2025-10-30

Este documento reúne una reflexión breve sobre las decisiones de diseño, los patrones de integración aplicados y los riesgos/limitaciones observados, junto con un espacio para evidencias (capturas) del funcionamiento.

1) ¿Qué patrón aplicaste en cada fase del flujo y por qué?

Ingesta (File → Camel): Patrón File Transfer / Polling Consumer.

El endpoint file:// lee de una carpeta de entrada (inbox) y detecta nuevos archivos. Este patrón simplifica la integración cuando los sistemas no están conectados directamente o requieren desacoplamiento temporal.

Preprocesamiento: Splitter + Message Translator.

Se descompone el CSV en filas (Splitter) y cada fila se transforma a un INSERT SQL (Translator). Esto separa responsabilidades: primero dividir el mensaje, luego traducir su contenido al formato que requiere el destino.

Orquestación interna: Pipes and Filters / Direct Channel.

Las etapas se encadenan con rutas Camel (pipes) y canales direct: para pasar el mensaje entre pasos sin acoplarse a detalles de implementación.

Persistencia (DB): JDBC Endpoint (Message Endpoint).

Se usa el componente jdbc con un DataSource. El endpoint encapsula la comunicación con MySQL, permitiendo cambiar parámetros de conexión sin modificar la lógica de transformación.

Post-procesamiento de archivo: Content-Based Router (implícito) + Move/Done.

Según el resultado, el archivo se mueve a outbox/ para dejar evidencia de que fue consumido. Este movimiento funciona como una señal de fin de flujo y evita reprocesamientos accidentales.

2) ¿Qué riesgos observas al usar una base de datos compartida?

- Acoplamiento fuerte entre aplicaciones (cambios de esquema rompen a todos).
- Contención y competencia por recursos (bloqueos, latencia y caída del rendimiento).
- Evolución de esquema difícil (versionado de columnas/tablas, migraciones coordinadas).
- Fugas de responsabilidades (cada app hace su propia lógica cerca de la base).
- Riesgos de seguridad y gobierno de datos (accesos excesivos, auditoría incompleta).
- Anomalías de concurrencia (lecturas sucias, phantom reads) si el aislamiento no está bien definido.

Buenas prácticas: aislar escritura por servicio, exponer datos a terceros vía APIs/Views controladas; usar migraciones versionadas; preferir CDC/eventos para integración; limitar privilegios y auditar accesos; usar réplicas de solo-lectura si procede.

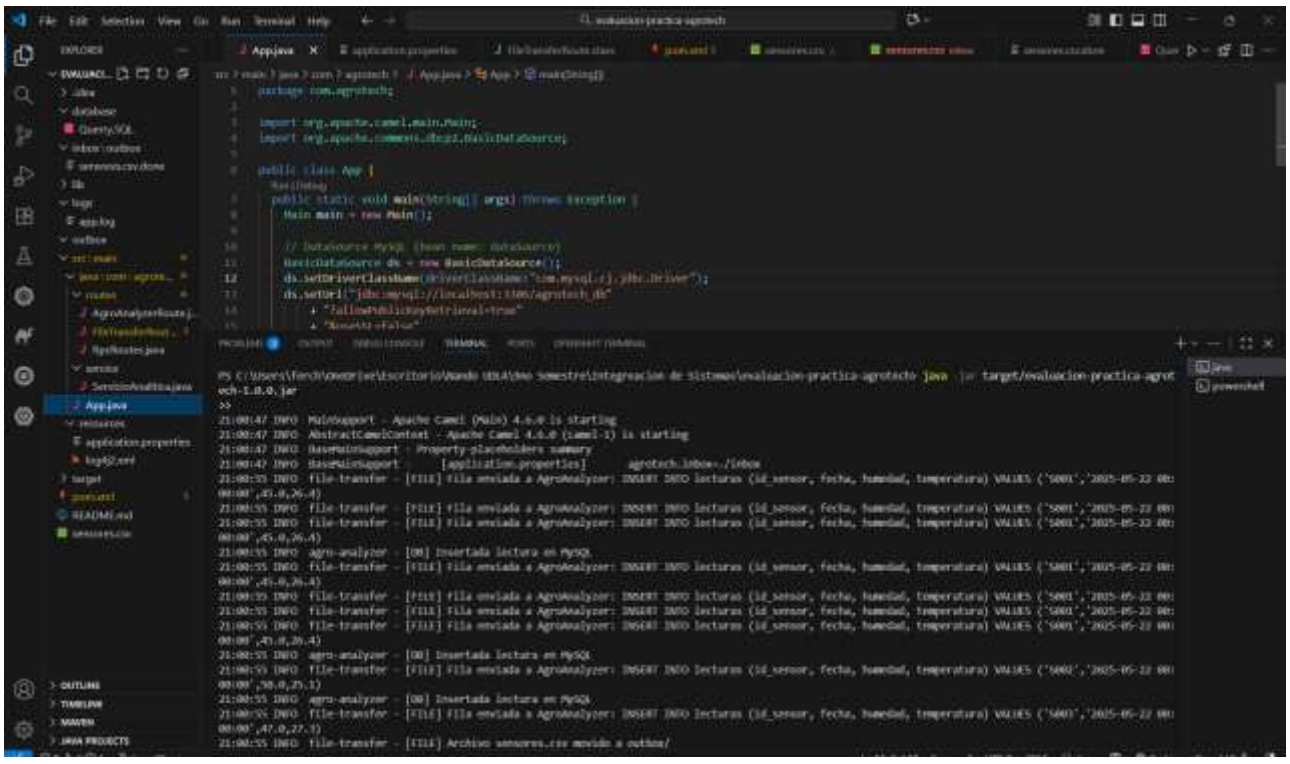
3) ¿Cómo ayuda el RPC simulado a representar un flujo síncrono?

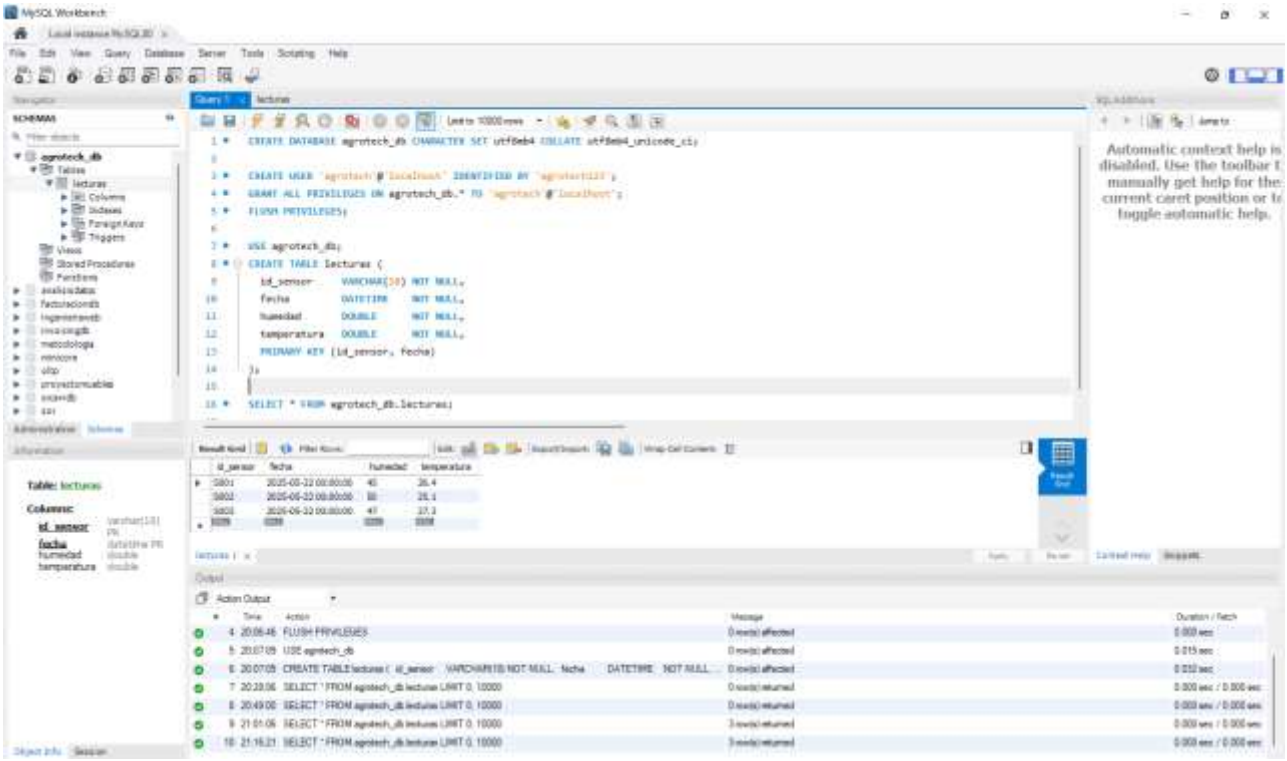
El RPC simulado con canales direct: modela un Request-Reply donde el cliente bloquea hasta recibir respuesta. Esto captura la semántica de sincronía (timeouts, propagación de errores, dependencia temporal) sin salir del proceso. Posteriormente, el mismo contrato puede portarse a HTTP/gRPC o mensajería con correlación si se requiere distribución.

4) ¿Qué limitaciones tienen los patrones clásicos frente a arquitecturas modernas?

- **Escalabilidad y elasticidad:** los patrones por archivo o RPC síncrono escalan peor que colas/eventos distribuidos.
- **Evolución de esquemas:** CSV/SQL acoplado es frágil comparado con contratos versionados (Avro/Protobuf/JSON Schema).
- **Resiliencia:** el backpressure y la tolerancia a fallos es superior en plataformas de streaming (Kafka/Pulsar) y colas robustas.
- Observabilidad: tracing y métricas finas requieren instrumentación extra; en EDA se integran mejor con trazas distribuidas.
- **Desacoplamiento organizacional:** equipos se mueven más rápido cuando integran por eventos/versionado que compartiendo DB.

5) Evidencias





Repositorio de Git-Hub: https://github.com/nandomartin7/Examen-Pr-ctico-Progreso-1--P2.git

6) Conclusión

Los EIP clásicos siguen siendo útiles dentro de un servicio o como puente temporal, pero para alta escala y evolución continua conviene migrar a APIs versionadas y/o integración basada en eventos (CDC, colas, streams).