**2. Funcionalidades que faltan (a nivel de negocio)**

1. **Asignación Operario↔Furgoneta “por día” aplicada a la lógica de consumo**
   * Existe la tabla asignaciones\_furgoneta(operario\_id, fecha, furgoneta\_id) pero **no se usa** aún.
   * Falta implementar una **función canónica**:
   * def resolver\_almacen\_operario(fecha: date, operario\_id: int) -> int:
   * # devuelve el almacen\_id asignado a ese operario ese día
   * Esta función debe utilizarse en **Imputación** y **Movimientos rápidos**, de modo que cualquier operación realizada por un operario afecte **a la furgoneta que tiene asignada ese día**.
2. **Pedido Ideal → Pedido real (ciclo completo)**
   * El cálculo está hecho, pero falta generar **pedidos reales** (pedidos + pedidos\_detalle) y exportarlos a **XLSX/CSV** agrupados por proveedor.
   * Estado de pedido: BORRADOR → ENVIADO → RECIBIDO.

*Es correcto, aunque habría que especificar el proceso que se seguirá. Debe permitir exportar ese borrador (PDF/EXCEL), después puedas confirmar el pedido modificando cantidades a lo que realmente se ha pedido, y el recibido tiene que ser automático, según van entrando albaranes. Los pedidos enviados, deben separarse por proveedor.*

1. **~~Auditoría de operaciones~~**
   * ~~Crear tabla auditoria con campos: id, usuario, acción, timestamp, payload.~~
   * ~~Añadir función audit(action, user, payload) llamada desde: Recepción, Traspasos, Imputación, Inventario, Anulación.~~
2. **Anulación de movimientos sin borrado**
   * En vez de eliminar registros, generar **contramovimiento vinculado al movimiento original** (movimiento\_id\_origen) y marcarlo como ANULADO.
   * Actualizar stock automáticamente.
3. **Coste de materiales (PEPS o CMP)**
   * Implementar cálculo de coste correcto por artículo, heredando el coste real de la entrada correspondiente.

*Debo estudiar bien si es mejor por PEPS o CMP. No son productos perecederos, y la variación de su precio, me la da el mercado. Si viene una remesa de importe mas bajo, debo amortiguar el golpe de los otros; pero si viene una partida mas cara, aunque pueda afrontar una pequeña parte compensando con los otros, debo actualizar mi precio al precio del mercado (ganando dinero con los que compre cuando eran baratos).*

1. **Bloqueo de almacenes en inventario**
   * Si un almacén tiene un inventario “abierto”, **no debe permitir movimientos** hasta que se cierre. *Cuando se empieza a hacer un inventario, no debe permitir hacer otra cosa hasta cerrar el inventario. Si se necesita hacer otra cosa, se cierra inventario este por donde este, pero no puede quedar un inventariado abierto. Es más, no debería permitir cambiar ventana a quien a empezado inventario, hasta que lo finalice: ya bien sea introduciendo todas las cantidades, o introduciendo solo lo que le haya dado tiempo, o cerrando directamente sin haber introducido nada.*
   * Añadir bandera inventario\_abierto o validación en inventarios.
2. **Exportaciones e informes de dirección**
   * Crear reporte consolidado en XLSX con:
     + Consumo semanal por operario *– Esto tendremos que debatir cual será el mejor modo de hacerlo, quiero poder sacar un informe con mucha información en poco espacio (que me quepa en un folio).*
     + Artículos más consumidos
     + Stock valorado total
     + OTs más caras
3. **Sistema de backups automáticos**
   * Script backup\_db.py con copias rotativas diarias del archivo almacen.db. *Mínimo 1 archivo diario, y duración de conservación 20 días. (al final, habrá solo 20 archivos de backups que se irán renovando día a día)*

**❗ 3. Errores estructurales que hay que corregir sí o sí**

1. **Esquemas duplicados en Furgonetas**
   * En db/schema.sql existe asignaciones\_furgoneta, pero en furgonetas\_repo.py hay otro bloque SCHEMA\_SQL que crea tablas distintas (furgonetas\_asignaciones con operario TEXT sin FK).
   * **Acción:**
     + Eliminar todos los CREATE TABLE embebidos en el código.
     + Mantener un solo esquema fuente (schema.sql).
     + Modificar los repositorios para trabajar con las tablas oficiales.
     + Si hay datos de prueba, crear un script de migración.
2. **SQL dentro de las ventanas (UI)**
   * Muchas ventanas usan sqlite3.connect() directamente y queries dentro del código de interfaz.
   * **Acción:** trasladar toda la lógica de acceso a datos a services/ o repos/.  
     Las ventanas deben llamar solo a funciones o métodos de servicio.
3. **Gestión de conexión y commits**
   * Algunas funciones aún no usan with get\_connection() as con:.
   * **Acción:** refactorizar para que todos los accesos usen context managers y commits automáticos.
4. **~~Hash de contraseñas sin sal~~**
   * ~~SHA-256 plano no es seguro.~~
   * **~~Acción:~~** ~~migrar a bcrypt o argon2.  
     Guardar salt y re-hash en login si detectas formato antiguo.~~
5. **Inventario sin control de bloqueo**
   * Movimientos sobre almacenes en inventario abierto generan inconsistencias.
   * **Acción:** impedirlo desde la capa de servicios.

*Esto creo que ya lo solucionamos en el punto 6 anterior.*

1. **UI con tamaños fijos**
   * Muchos formularios usan setFixedSize().
   * **Acción:** usar layouts responsivos y resize() para mejorar compatibilidad con pantallas distintas.
2. **Logging básico**
   * El log app.log no guarda usuario ni acción.
   * **Acción:** añadir logs detallados con usuario, ventana y tipo de operación (INFO/WARN/ERROR).
3. **Migraciones de esquema**
   * Actualmente solo hay schema.sql.
   * **Acción:** crear scripts/migrate.py con un sistema de versiones (schema\_version en DB) para aplicar actualizaciones incrementales sin perder datos.

**⚡ 4. Mejoras de alto impacto (rápidas y útiles)**

* Implementar resolver\_almacen\_operario() en services/operativa\_service.py.
* Llamar a esta función desde las operaciones de **Imputación** y **Movimiento**.
* Crear botón “**Asignar hoy**” en menú principal para asignar Operario→Furgoneta del día. *Se debería poder asignar desde Movimientos directamente.*
* Agregar control de “desajustes” (operarios sin furgoneta asignada hoy).
* Añadir **anulación estándar**: crea contramovimiento + registro en auditoría. *(Solo debe poder hacer esas anulaciones Admin)*
* Exportar **Pedidos** a XLSX/PDF por proveedor.
* Implementar **script de backup diario** de la DB.
* ~~Crear~~ **~~tests mínimos~~** ~~(pytest) para:~~
  + ~~resolver\_almacen\_operario()~~
  + ~~contramovimientos~~
  + ~~pedido ideal~~
  + ~~bloqueo de inventario~~

**🚀 PROPUESTAS DE MEJORA Y EVOLUCIÓN**

**1. 🔄 Arquitectura y escalabilidad**

**✅ Cambiar:**

* **De “monolito PySide6” a arquitectura modular con servicios internos.**  
  Divide el código en 3 niveles:
  + /core → lógica común (db\_utils, auth, settings)
  + /domain → reglas de negocio (recepción, imputación, traspasos, inventarios)
  + /ui → vistas y controladores (Qt).  
    Esto te permitirá, en el futuro, sustituir la UI por una web o API sin tocar la lógica.

**💡 Añadir:**

* **API REST local (FastAPI o Flask mínimo)** para servir datos a otras herramientas o apps móviles.  
  Ejemplo: /api/articulos, /api/stock, /api/movimientos.  
  Serviría para:
  + Panel web de reportes
  + App Android de operarios (lectura QR, imputar material, etc.)
  + Sincronización con Odoo o Power BI

**2. 🧱 Base de datos y consistencia**

**✅ Cambiar:**

* Migrar gradualmente de SQLite → **PostgreSQL**, manteniendo compatibilidad mediante capa db\_utils.  
  SQLite es genial para entorno local, pero si quieres que varias personas trabajen a la vez, necesitas Postgres.

**💡 Añadir:**

* **Triggers SQL** para validaciones automáticas (ej. impedir movimientos si stock<0).
* **Histórico de precios y proveedores**: tabla precios\_proveedor(id\_articulo, proveedor, precio, fecha\_vigencia)  
  Así podrás analizar tendencias y revalorizar stock antiguo.

**3. 🔐 Seguridad y auditoría**

**💡 Añadir:**

* **Control de permisos a nivel de acción**, no solo de ventana.  
  Ejemplo:
  + “Operario” puede imputar pero no borrar.
  + “Responsable de almacén” puede anular movimientos.
  + “Administrador” puede hacer inventarios globales.
* **Historial completo de cada artículo:**  
  Cada acción (entrada, salida, devolución, imputación) visible en una línea temporal.  
  Algo tipo:
* 2025-10-28 | Recepción | Albarán 341 | +2 uds | proveedor BAXI
* 2025-10-29 | Imputación | OT123 | -1 ud | operario Daniel
* 2025-10-30 | Devolución | Furgoneta 3 | +1 ud

**4. 🧮 Inteligencia operativa**

**💡 Añadir:**

* **Previsión de consumo semanal por operario o categoría**, basada en histórico de los últimos 3 meses.  
  (Pequeño modelo predictivo o media ponderada).  
  → Útil para reponer stock antes de quedarse sin material.
* **Indicadores de alerta (Dashboard):**
  + Artículos por debajo del stock mínimo.
  + Furgonetas con consumos anómalos.
  + Material imputado a OT sin cerrar.
* **Valoración de stock total** con coste medio, para informes contables.

**5. 🧰 Flujo de trabajo operativo**

**💡 Añadir:**

* **Control de entregas parciales:**  
  Si un pedido llega incompleto, registrar las líneas faltantes para seguimiento automático.
* **Módulo de incidencias** vinculado a movimientos (rotura, material erróneo, devolución al proveedor con motivo, etc.).
* **Asignación automática de herramientas personales:**  
  Cada operario con su inventario propio (herramientas fijas), distinto del material consumible.
* **Modo offline:**  
  Si algún día usas tablet o app móvil, el programa podría trabajar en local y sincronizar luego.

**6. 🎨 Interfaz y experiencia de usuario**

**✅ Cambiar:**

* Sustituir tablas estáticas por QTableView con **modelos dinámicos** y búsqueda instantánea (QSortFilterProxyModel).  
  Mucho más fluido con grandes volúmenes de datos.
* Crear una **barra lateral tipo “panel de control”**, con iconos de módulos y resumen rápido:
  + Stock total
  + Movimientos del día
  + Pendientes de imputar
  + Furgonetas activas

**💡 Añadir:**

* **Modo oscuro / claro** configurable.
* **Sistema de notificaciones internas** (por ejemplo: “Stock crítico en Furgoneta 2”).
* **Atajos globales** (Ctrl+N = nuevo movimiento, F2 = buscador, etc.).

**7. 📊 Reportes y gestión avanzada**

**💡 Añadir:**

* **Reportes automáticos programados** (semana/mes) exportados en /exports:
  + Consumo por operario y coste total
  + Stock final de cada furgoneta
  + Entradas y devoluciones por proveedor
  + Valor de stock total
* **Integración con Power BI o Looker Studio** (usando el API REST o exportación CSV automática).

**8. ⚙️ Mantenimiento y fiabilidad**

**💡 Añadir:**

* **Sistema de logs rotativos** (logging.handlers.RotatingFileHandler) para que app.log no crezca infinito.
* **Monitor de rendimiento**: loguear tiempos de cada operación SQL (para optimizar consultas).
* **Validación de integridad de DB**: script semanal que revise claves, duplicados, stock negativo.
* **Sistema de backup automático con hash de integridad SHA256** (para detectar archivos alterados).

**9. 🧪 Calidad de código**

**💡 Añadir:**

* **Tests unitarios y de integración** con pytest para los módulos de negocio (recepción, traspasos, imputaciones).
* **Script de “pre-lanzamiento”** (make build o python manage.py check) que valide dependencias, schema y versiones.
* **Configuración por entorno** (dev/test/prod) en config.ini.

**10. 🧭 Visión a futuro**

**🌐 Odoo-ready**

El paso natural será que este programa sea el **cliente ligero de Odoo Community**, usando su API RPC o XML-RPC.  
El objetivo:

* Que tu programa siga siendo el **frontend ágil** para operarios/almacén.
* Y que Odoo centralice la contabilidad, CRM, facturación, etc.