Asunto: Decisión de Arquitectura: Adopción de Monorepo Organizado
Fecha: 17 de noviembre de 2025
Participantes: Pedro J (Desarrollador), Gemini (Consultor de Arquitectura)

1. Resumen de la Decisión

Tras analizar los trade-offs de la arquitectura monolítica actual, se ha tomado la decisión estratégica de **refactorizar el proyecto "Ortiz y Cía. Consignatarios" a una arquitectura de Monorepo Organizado.**
Esta decisión implica mantener un **único repositorio en GitHub**, pero reestructurando las carpetas internas para desacoplar lógicamente la web_app (el servidor Flask) del data_pipeline (el scraper y generador de reportes).

2. Contexto del Debate

La arquitectura monolítica actual presenta los siguientes desafíos:

- **Despliegue Pesado:** La aplicación web (app.py) se ve forzada a instalar dependencias que no utiliza.
- **Acoplamiento:** El código de la web y el del pipeline están mezclados, dificultando el mantenimiento y la identificación de responsabilidades.
- **Escalabilidad Rígida:** Es imposible escalar la aplicación web sin escalar también el pipeline de datos.

Se descartó la alternativa de "Dos Repositorios Separados" debido al alto riesgo de desincronización del código compartido (específicamente db_manager.py), lo que podría introducir *bugs* críticos.

3. Arquitectura Seleccionada: Monorepo Organizado

El Monorepo Organizado ("Después") se selecciona por ser la arquitectura que ofrece el mejor balance de beneficios para este proyecto:

- ✅ **Cero Duplicación de Código:** Mantenemos un solo db_manager.py en una carpeta shared_code/, que es importada por ambas aplicaciones. Esto elimina el riesgo de desincronización.
- ✅ **Despliegues Ligeros e Independientes:** Cada aplicación (web_app y data_pipeline) tendrá su propio archivo requirements.txt. Al desplegar, solo se instalarán las dependencias necesarias para cada servicio.
- ✅ **Commits Atómicos:** Los cambios en el código compartido (shared_code/) y en las aplicaciones que lo consumen (web_app/) se pueden realizar en un **único commit**, garantizando la integridad del repositorio en todo momento.
- ❌ **Único Trade-off:** La configuración inicial es ligeramente más compleja, ya que requiere modificar el sys.path en app.py y main.py para que puedan "ver" e importar la carpeta shared_code/. Este es un costo de configuración aceptable a cambio de la robustez a largo plazo.

4. Diagramas de Arquitectura

A continuación, se presentan los diagramas "Antes" y "Después" de la arquitectura del sistema.

Diagrama 1: Arquitectura "Antes" (Monolito Acoplado)

Este diagrama muestra cómo ambas aplicaciones comparten un mismo entorno y requirements.txt, y cómo las plantillas web y de PDF están mezcladas.

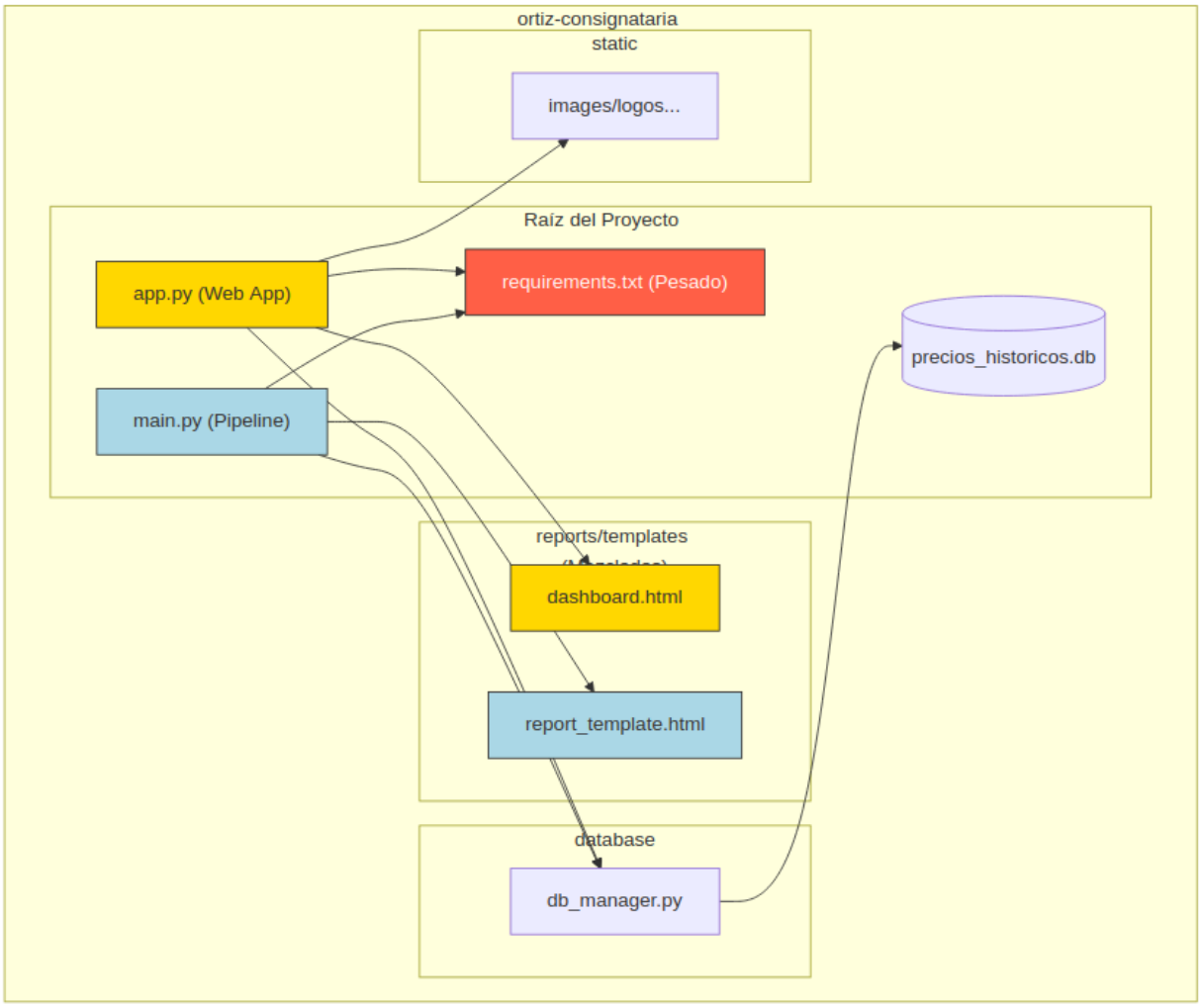
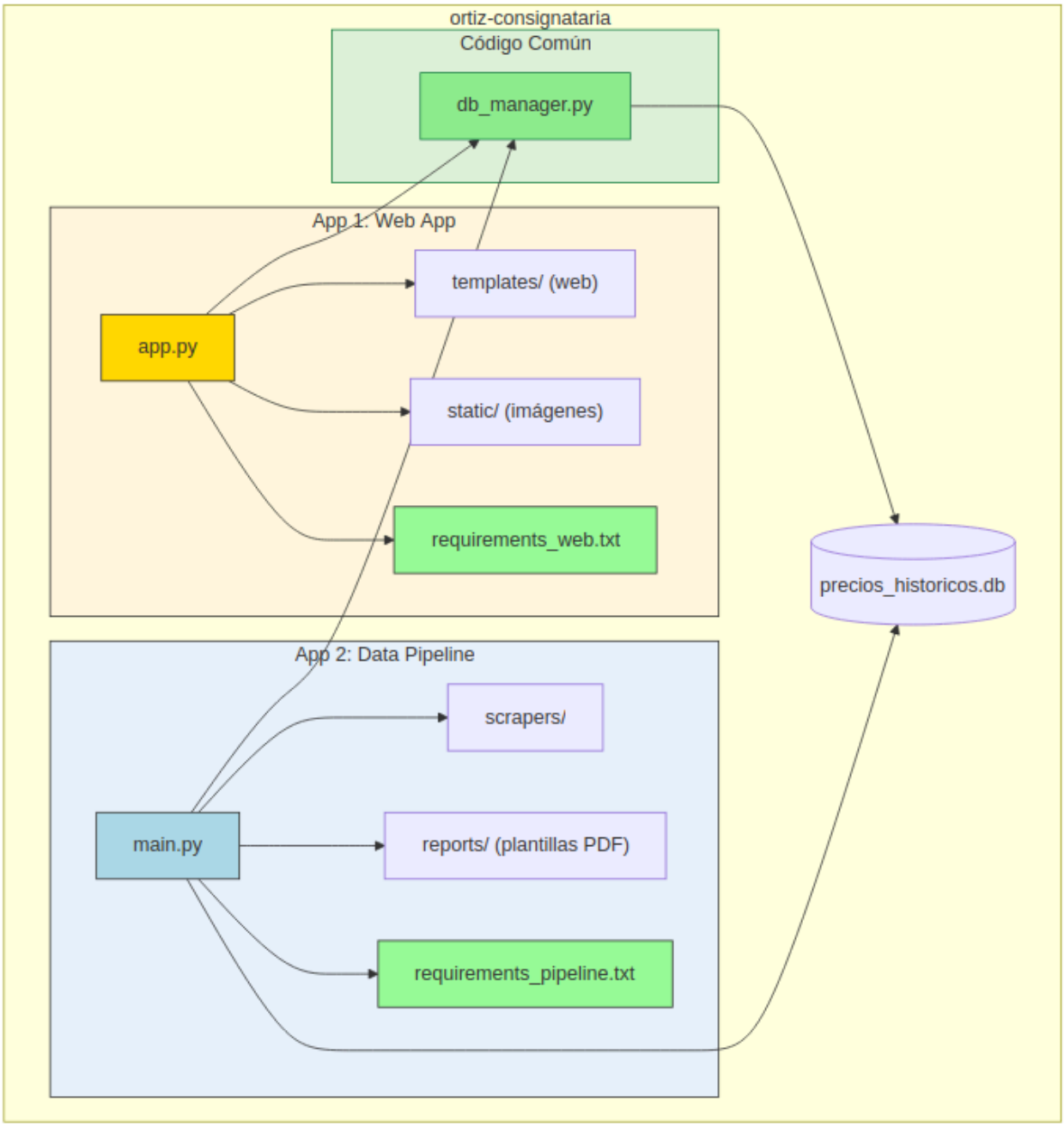


Diagrama 2: Arquitectura "Después" (Monorepo Organizado)

Este diagrama muestra las dos aplicaciones desacopladas, cada una con sus propias dependencias, importando desde una carpeta de código compartido.



5. Próximos Pasos (Accionables)

Para implementar esta nueva arquitectura, se deben seguir los siguientes pasos:

1. **Crear Carpetas:** Crear las carpetas web_app, data_pipeline, y shared_code en la raíz del repositorio.
2. **Mover Archivos:**
    - Mover app.py, templates/ (con base.html, inicio.html, dashboard.html) y static/ dentro de web_app/.

    - Mover main.py, scrapers/, reports/, utils/ (y otros scripts de pipeline) dentro de data_pipeline/.
    - Mover la carpeta database/ (conteniendo db_manager.py) dentro de shared_code/.
3. **Actualizar sys.path:** Modificar web_app/app.py y data_pipeline/main.py para que añadan la raíz del proyecto al sys.path, permitiéndoles importar desde shared_code/.
4. **Separar Dependencias:** Crear requirements_web.txt (con Flask, Flask-Cors) y requirements_pipeline.txt (con requests, weasyprint, etc.) y eliminar el requirements.txt raíz.
5. **Ajustar .gitignore:** Asegurarse de que precios_historicos.db (que queda en la raíz) esté en el .gitignore.