ApplicationContext y Orchestrator: manejan inyección de dependencias/ciclo de vida controlado (ahondar en esto).

Flujo visualmente claro: repo -> servicio -> controlador -> UI

🧱 1. Arquitectura y Dependencias

1. **⚠️ Lo mejorable**
2. **Exceso de responsabilidades en la capa service  
   Muchos servicios mezclan lógica de negocio, conversión de DTOs, validación y coordinación entre repositorios.  
   👉 *Posible mejora:* separar el mapeo DTOs ↔ entidades en una capa de *mappers* o *adapters* para aislar reglas de negocio reales.**
3. **Dependencia vertical muy rígida  
   Si los servicios conocen directamente a los repositorios concretos, se reduce la intercambiabilidad y dificulta pruebas unitarias.  
   👉 *Solución:* introducir interfaces abstractas (protocols) y registrar implementaciones concretas en ApplicationContext.**
4. **Repositorios con validación  
   Varios repositorios tienen \_validate\_related\_objects(), \_validate\_audit\_data(), etc.  
   Estas validaciones probablemente deberían moverse a:**
   * **una capa de Domain Service, o**
   * **validadores reutilizables fuera de los repositorios.  
     👉 *Así desacoplas acceso a datos de lógica de negocio.***
5. **Globals o “puntos de entrada difusos”  
   Los nodos (global)() en muchos módulos indican que hay código ejecutándose al importar el archivo (no encapsulado en funciones o clases).  
   👉 *Eso puede generar acoplamiento temporal (orden de importación importa).*  
   Centraliza toda inicialización en ApplicationOrchestrator o ApplicationContext.**
6. **⚙️ 2. Organización y Escalabilidad**
7. **Patrón repetitivo: create, update, get\_by\_\***  
   Si cada repositorio lo redefine, podrías:
   * Consolidar lógica común en BaseRepository.
   * Usar **métodos genéricos parametrizados** (por ejemplo, con SQLAlchemy y tipos de modelo).  
     👉 Evitarás repetición y podrás mejorar la trazabilidad de errores.
8. **Servicios con métodos duplicados**  
   Ejemplo: get\_all\_bugs\_service\_dto, get\_all\_bugs\_for\_table, etc.  
   👉 Define un patrón de *query handlers* o *use cases* independientes:  
   GetAllBugsHandler, UpdateBugFromFormHandler, etc.  
   Esto separa casos de uso de lógica de infraestructura.
9. **Controladores tabulares (UI)**  
   Los TabController parecen encargarse de la interacción UI ↔ servicios.  
   Si manejan mucha lógica de datos, podrías mover esa parte a un **ViewModel** (patrón MVVM), manteniendo el controlador centrado en eventos UI.
10. **Refactor de ApplicationContext**
    * Evalúa si estás usando un patrón *Singleton* con get\_instance() (lo parece).
    * Podrías reemplazarlo por un **contenedor de dependencias explícito** (como *Dependency Injector* o *wired*), para mejorar testabilidad.

| **Tipo** | **Recomendación** | **Beneficio** |
| --- | --- | --- |
| **Performance** | Implementar *lazy loading* en repositorios con relaciones complejas (get\_with\_relations) | Menor coste en consultas |
| **Logging / Tracing** | Usa middleware o decoradores para instrumentar operaciones de servicio y repositorio | Facilita debugging y auditoría |
| **Error handling** | Define excepciones propias (DomainError, RepositoryError, etc.) y centraliza su captura | Aísla errores por capa |
| **Configuración** | Extraer rutas, nombres de tablas, etc. a config.yaml o pydantic.BaseSettings | Desacopla del código |
| **Tests** | Inyectar repos fake/mocks en servicios | Mejora la cobertura sin tocar DB real |
| **DTOs** | Reemplazar dataclasses manuales por pydantic (si no lo usas aún) | Validación y serialización automáticas |

1. **🧩 4. Refactor estratégico sugerido**
2. **Identificar capas puras:**
   * repository → infraestructura
   * service → aplicación
   * controller → presentación
3. **Aislar Domain:**
   * Extrae lógica de negocio y validación de entidades del repositorio.
   * Define modelos de dominio independientes del ORM.
4. **Eliminar dependencias cíclicas:**
   * Evita que un service importe a otro service directamente.
   * Usa *event dispatchers* o *mediator pattern* si necesitas comunicación transversal.
5. **Simplificar inicialización:**
   * Delega inicialización compleja (initialize\_services, \_initialize\_database) a un “bootstrap” modular.
   * Mantén ApplicationContext libre de dependencias de UI o infraestructura.

| **Prioridad** | **Acción** | **Resultado esperado** |
| --- | --- | --- |
| Alta | Extraer validaciones y mapeos DTO a capas dedicadas | Código más limpio y testeable |
| Alta | Crear interfaces (protocols) para repositorios/servicios | Facilita mocks y tests |
| Media | Modularizar ApplicationContext (inyección explícita) | Menor acoplamiento global |
| Media | Consolidar base común de CRUD (herencia o mixin genérico) | Menos duplicación |
| Baja | Adoptar patrón Mediator/EventBus entre servicios | Mejor comunicación desacoplada |
| Baja | Añadir análisis estático (mypy, pylint, code2flow -D) | Detección temprana de dependencias cíclicas |

1. **🔮 Bonus: próxima evolución posible**

Si quisieras **llevar esta arquitectura al siguiente nivel**, podrías:

* Adoptar un enfoque **“hexagonal” (ports & adapters)**:  
  servicios definen puertos, repositorios son adaptadores.
* Implementar **Use Case classes** en lugar de métodos dentro del servicio.
* Automatizar parte del mapeo DTO ↔ entidad con *pydantic models* o *SQLModel*.
* Añadir **test de integración capa a capa** (repository → service → controller) para asegurar aislamiento real.
  1. **🧩 FASE 1 — Aislar dependencias y estabilizar la arquitectura**

**Objetivo:** poder refactorizar sin romper nada.  
**Duración típica:** 1–2 sprints cortos.

# **Centralizar inicialización (bootstrap)**

**Por qué:** todo lo demás depende de tener control sobre cuándo y cómo se crean servicios/repositorios.  
**Qué hacer:**

* Revisa ApplicationContext y ApplicationOrchestrator: que sean los **únicos puntos** que inicializan repositorios, servicios y controladores.
* Asegúrate de que **no haya código ejecutándose en imports** ((global)() → muévelo a funciones explícitas).
* Añade un método bootstrap() que inicialice todo el entorno de forma determinista.

👉 Resultado: puedes levantar o testear cualquier parte del sistema sin efectos colaterales.

1. **Introducir inyección de dependencias (aunque sea manual)**

**Por qué:** necesitas aislar los servicios de los repositorios concretos antes de tocar su código.

**Qué hacer:**

* Empieza por definir **interfaces abstractas (protocols)** para repositorios.

class IBugRepository(Protocol):

def create(...): ...

def get\_by\_id(...): ...

En ApplicationContext, registra implementaciones concretas (BugRepository, etc.).

Cambia servicios para depender de las interfaces, no de clases reales.

Resultado: el resto de capas no depende de implementaciones concretas.  
Esto te abre la puerta a test unitarios sin DB y a refactors seguros.

1. **⚙️ FASE 2 — Limpiar responsabilidades**

**Objetivo:** reducir ruido y duplicación sin alterar comportamiento visible.  
**Duración:** 2–4 sprints según el tamaño.

1. **3️⃣ Extraer validaciones de los repositorios**

**Por qué:** las validaciones en \_validate\_\* mezclan persistencia con dominio.  
**Qué hacer:**

* Crea un módulo validators/ con funciones puras que reciban entidades o DTOs.
* Sustituye las llamadas dentro de repositorios por llamadas al validador.

👉 Resultado: repositorios limpios, validaciones testeables.

1. **4️⃣ Extraer mapeos DTO ↔ entidades a “mappers”**

**Por qué:** actualmente el service hace demasiadas cosas (negocio + transformación).  
**Qué hacer:**

* Crea un módulo mappers/ con clases tipo:

class BugMapper:

def to\_dto(entity: Bug) -> BugDTO: ...

def from\_form(form: BugFormDTO) -> Bug: ...

* Llama a estos mapeadores desde los servicios.

👉 Resultado: los servicios ahora solo orquestan y aplican reglas, no transforman datos.

1. **5️⃣ Consolidar métodos genéricos en las bases (BaseRepository, BaseService)**

**Por qué:** ahora que tienes interfaces y validaciones externas, puedes unificar lo repetido.  
**Qué hacer:**

* Mueve CRUDs repetidos (create, update, get\_by\_id, etc.) a las clases base.
* Solo deja lógica específica del dominio en los repositorios concretos.

👉 Resultado: menos código duplicado, menos mantenimiento.

1. **🧱 FASE 3 — Optimizar la comunicación entre capas**

**Objetivo:** limpiar el flujo vertical y eliminar acoplamientos indirectos.  
**Duración:** 1–2 sprints.

1. **6️⃣ Desacoplar servicios entre sí**

**Por qué:** algunos servicios probablemente se llaman entre ellos directamente (get\_service()), generando dependencias cruzadas.  
**Qué hacer:**

* Introduce un **Event Bus o Mediator interno** (puede ser una simple clase EventDispatcher con subscribe/emit).
* Sustituye llamadas directas entre servicios por eventos (p. ej. “bug\_created” → BugService emite, otro escucha).

👉 Resultado: servicios independientes y más fáciles de testear.

1. **7️⃣ Simplificar ApplicationContext**

**Por qué:** tras mover responsabilidades y añadir DI, puede quedar hinchado.  
**Qué hacer:**

* Divide su lógica en módulos: bootstrap\_services.py, bootstrap\_ui.py, etc.
* Convierte ApplicationContext en un *contenedor simple de dependencias*, no un coordinador.

👉 Resultado: orquestación más clara y mínima.

1. **🧠 FASE 4 — Evolución y escalabilidad**

**Objetivo:** preparar el sistema para crecer o integrar nuevas features.  
**Duración:** indefinida (mejoras evolutivas).

1. **8️⃣ Introducir “Use Cases” (Command/Query)**

**Por qué:** ahora que cada capa está limpia, puedes encapsular la lógica de negocio real.  
**Qué hacer:**

* Divide los servicios grandes (BugService, RequirementService) en comandos o casos de uso:

use\_cases/

├── create\_bug.py

├── update\_requirement.py

└── list\_bugs\_for\_table.py

1. **9️⃣ Adopta tipado y validación declarativa**

**Por qué:** ahora puedes aplicar herramientas más seguras sin miedo a romper la estructura.  
**Qué hacer:**

* Añade mypy o pydantic a DTOs.
* Tipar interfaces y métodos del dominio.

👉 Resultado: seguridad estática, detección temprana de errores.

1. **🔟 Refactor UI / Controllers hacia MVVM o MVP**

**Por qué:** los TabController parecen contener demasiada lógica.  
**Qué hacer:**

* Introduce *ViewModels* que gestionen estado y exposición de datos.
* Deja los controladores solo para eventos (clicks, actualizaciones, etc.).

👉 Resultado: separación clara entre lógica de presentación y estado.