1. DISE;O Y ARQUITECTURA DE SOFTWARE

* Auth Server separado: emite JWT corto + refresh con rotación; contraseñas con Argo2id; MFA opcional.
* RBAC + OPA: roles para lo general, políticas (ABAC) para lo fino sin meter reglas en cada servicio.
* Eventos: todo lo importante (login, asignación de rol, denegaciones) se publica y la auditoría lo procesa sin frenar al usuario.
* Auditoría CQRS: escritura append-only en storage barato y búsquedas en Elastic.
* DB por servicio y caché en Redis para grants, control y politicas.
* Observabilidad desde el día 1 (logs, métricas, etc).

**Flujo de datos**

* Login: cliente/ -> Auth -> valida (MFA si aplica) -> emite JWT -> evento Auth.Login -> se indexa en auditoría.
* Autorización: servicio recibe JWT -> pide a Policy Service (subject, action, resource) -> allow/deny ->se registra evento.
* Roles (admin): asignar/elimina roles en Role Service -> publica evento -> auditoría lo guarda; Policy Service invalida caché
* Consulta audit: la consola admin busca por fecha/actor/acción.

**Endpoints**

* Auth:
  + POST /oauth2/token,
  + POST /oauth2/refresh,
  + POST /oauth2/introspect
* Users:
  + POST /users,
  + GET /users/{id},
  + PATCH /users/{id} (activar/bloquear)
* Roles:
  + POST /roles/{roleId}/assign/{userId},
  + DELETE /roles/{roleId}/assign/{userId}
* Auth:
  + POST /authz/check {subject, action, resource, context, etc}
* Auditoría:
  + GET /audit/search?q=&from=&to=

3. CODE REVIEW (PYHON)

**Errores**

* Concatenación de SQL con f-strings: abierto a inyección SQL (... WHERE id = {user\_id}).
* Duplicados: get\_user\_data y get\_user\_data\_again hacen exactamente lo mismo.
* Manejo de errores pobre: no hay try/except; si no hay filas, fetchone() devuelve None y el caller no distingue entre no encontrado y el error real.
* Gestión de recursos: no usa context managers, abre/cierra la conexión en cada llamada y no configura row\_factory.
* Typing poco claro, retorna None sin documentarlo.
* Además usa SELECT \* (cambios de esquema).
* Faltan docstrings y validación de tipos de entrada.

**Propuesta**

* Query parametrizado: WHERE id = ? con parámetros
* Unificar funciones en una sola y regresar el tipo (User o None) o levantar una excepción NotFoundError.
* Usar context managers: with sqlite3.connect(...) as conn y conn.row\_factory = sqlite3.row.
* Evitar \* en el select ya que puede especificar columnas.
* Typing + docstring y manejo explícito del caso no encontrado.

4. DESEMPEÑO BAJO PRESION

**Falla en producción: “No se puede guardar, API devuelve 500”**

Hipótesis:

* Deploy/migración reciente.
* DB caída/saturada/o con bloqueos.
* Dependencia o servicios de externos caído.
* Validación de datos rompió el flujo.

Diagnóstico:

* Rollback o desactivar feature reciente.
* Revisar logs/stacktrace del endpoint (últimos 20 min).
* Revisar la DB: conexiones, locks, espacio, migraciones, USP.
* Reproducir con payload mínimo (prod vs local).
* Checar dependencias o servicios y sus timeouts.

Solucion:

* Rollback inmediato si fue el cambio.
* Aumentar pool/instancias si es saturación.
* Circuit breaker/retry o desactivar integración externa.

Comunicación:

* Aviso: “Hay un error en los ultimos minutos, se reviso el problema y se espera que funcione durante los siguientes minutos, mientras se sigue revisando el porque ocurrio”.
* Se continua con la revision del diagnostico para saber realmente que paso, con cada uno, si fue error interno o externo.

Saludos **:)**