GESCOMPH: Un Monolito Modular para Modernizar la Gestión Contractual en el Sector Público

Brayan Santiago Guerrero Mendez
SENA
Niva, Colombia
brayans_guerrero@soy.sena.edu.co

Resumen

Cualquiera que haya desarrollado software para el sector público conoce bien el problema: sistemas obsoletos, deuda técnica por todos lados y la sensación de que siempre estás apagando incendios. Con GESCOMPH queríamos romper ese ciclo. Este artículo presenta un sistema de gestión contractual que va contra la corriente de moda: no usamos microservicios. En su lugar, construimos un Monolito Modular con .NET 8 y SQL Server, algo que muchos dirían que está pasado de moda. Pero funcionó. La arquitectura nos dio lo mejor de ambos mundos: podemos desplegar todo de un tirón (nada de coordinar 20 servicios diferentes) y al mismo tiempo mantener los módulos bien separados gracias a Clean Architecture. A lo largo del artículo explicamos cómo implementamos la seguridad, incluyendo un sistema de rotación de tokens que nos dio más de un dolor de cabeza, las optimizaciones que hicimos con Entity Framework para no matar la base de datos, y cómo logramos que los costos en la nube no se disparen. Al final, los números nos dieron la razón: se puede tener un sistema mantenible sin caer en la complejidad innecesaria de los sistemas distribuidos. Creemos que esto puede servir como referencia práctica para otros proyectos de modernización en el gobierno.

Keywords

buenas prácticas, ingeniería de software, investigación aplicada, reproducibilidad

1. Introducción

Modernizar la tecnología en el sector público es mucho más que actualizar software; es un requisito para que el Estado responda mejor a los ciudadanos. El problema es que muchas instituciones siguen atadas a sistemas viejos, monolíticos y difíciles de mantener, que frenan cualquier intento de innovación.

Hoy en día, la tendencia casi automática es pensar que la solución a todo esto son los *microservicios*. Se venden como la arquitectura ideal para escalar y ser ágiles. Pero hay que tener cuidado. La literatura técnica y la experiencia práctica nos dicen otra cosa: romper una aplicación en mil pedazos antes de tiempo suele traer más dolores de cabeza que beneficios, especialmente si no se tiene un equipo enorme para gestionar esa complejidad.

Aquí es donde entra el **monolito modular**. No es un paso atrás, sino una decisión inteligente para sistemas de tamaño medio. La idea es simple: mantienes la simplicidad de tener todo en un solo lugar (sin latencia de red, despliegues fáciles), pero por dentro organizas el código con la misma disciplina que si fueran microservicios. Así obtienes lo mejor de los dos mundos.

Con esta mentalidad construimos **GESCOMPH**, un sistema para gestionar contratos públicos. No nos lanzamos a ciegas a usar lo último de moda. Preferimos usar *Clean Architecture* para que el negocio no dependa de la tecnología de turno. Esto nos permite mantener el código limpio y, si el día de mañana el sistema crece muchísimo, migrar a microservicios será mucho más fácil porque la casa ya está ordenada.

Para la seguridad, tampoco reinventamos la rueda: usamos estándares probados como **JWT** y **OAuth 2.0**, y todo corre sobre contenedores para que sea fácil de mover y escalar.

En este artículo no solo vamos a mostrar cómo está hecho `gescomph-api`. Queremos usarlo de ejemplo para discutir por qué, a veces, una arquitectura bien pensada y .ªburrida.es mucho mejor que una arquitectura compleja y "moderna". Vamos a ver datos de rendimiento y seguridad que respaldan por qué elegimos este camino.

2. Marco teórico y trabajos relacionados

La ingeniería de software vive en un péndulo constante. Hace unos años, todo era consolidar; hoy, la norma parece ser distribuir. Sin embargo, adoptar microservicios solo porque .es lo moderno.es una receta para el desastre. En esta sección, no vamos a hacer un listado aburrido de papers; vamos a analizar qué dice realmente la evidencia sobre cuándo y cómo conviene romper un sistema, y cómo eso justifica la arquitectura de GESCOMPH.

2.1. El dilema: ¿Monolito o Microservicios?

Es fácil dejarse llevar por el éxito de gigantes como Netflix, cuyo viaje hacia los microservicios está muy bien documentado [15]. Pero hay que leer la letra pequeña. Mazzara et al. [10] y Blinowski [2] ponen los pies en la tierra: si bien ganas escalabilidad, el precio a pagar en complejidad operativa es altísimo. No es una mejora automática; es un intercambio de problemas.

Figura 1: Comparativa conceptual entre Monolito y Microservicios. Se ilustra la complejidad operativa frente a la simplicidad de despliegue.

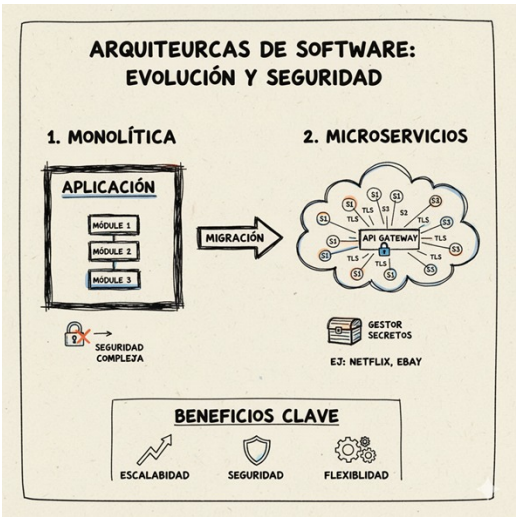

Figura 2: Modelado de dominios en arquitecturas modernas. La estructura del código debe reflejar los límites del negocio (DDD).

Para un proyecto académico como GESCOMPH, diseñado para explorar arquitecturas pragmáticas en contextos gubernamentales, la literatura sugiere prudencia. Velepucha y Flores [19] advierten sobre los desafíos de migrar sin una estrategia clara. Por eso, nuestra apuesta por un diseño modular no es timidez, es una decisión pedagógica: demostrar que se puede preparar el terreno para escalar sin adoptar prematuramente la complejidad de sistemas distribuidos.

2.2. No es solo código, es diseño

De nada sirve tener microservicios si por dentro son un desastre. Nivedhaa [12] y Oyeniran [14] recalcan que la arquitectura debe evolucionar con el negocio, no al revés. Aquí es donde brilla el *Domain-Driven Design* (DDD). Myllynen et al. [11] muestran cómo modelar los límites del sistema basándose en el dominio real y no en tablas de base de datos es lo que realmente permite desacoplar componentes.

En GESCOMPH, tomamos esto muy en serio. No separamos clases por capricho técnico, sino siguiendo los límites del negocio de gestión contractual.

2.3. Seguridad: Más piezas, más problemas

Cuando rompes un monolito, multiplicas tu superficie de ataque. De repente, llamadas que eran internas ahora viajan por la red. Billawa et al. [1] lo dejan claro: la seguridad en sistemas distribuidos no puede ser un pensamiento secundario. Requiere una gestión de identidad robusta y observabilidad total.

No basta con poner un firewall. Kothapalli [7] y Cruz-Cunha [3] insisten en la defensa en profundidad. Incluso se habla ya de esiliencia cibernética predictivaüsando IA [6], aunque para nuestro alcance actual, nos centramos en lo fundamental y efectivo: una implementación sólida de JWT y RBAC que no deje brechas, siguiendo las recomendaciones de Patlolla [16] sobre monitoreo constante.

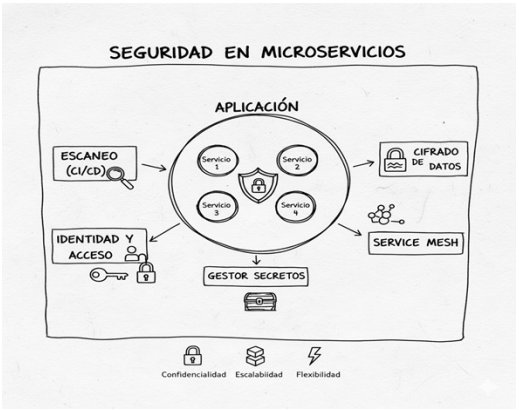

Figura 3: Desafíos de seguridad en entornos distribuidos. La superficie de ataque aumenta exponencialmente con la fragmentación de servicios.

2.4. La deuda técnica que no se ve

A veces, la peor deuda no es el código feo, es la mala arquitectura. Toledo et al. [18] explican cómo las decisiones tomadas bajo presión se calcifican y hacen que el sistema sea imposible de mantener años después.

GESCOMPH nace con la obsesión de evitar esto. Al usar *Clean Architecture*, estamos pagando una "prima de seguroïnicial: escribimos más código ahora y separamos más capas de las que parecen necesarias, para que en el futuro, cuando cambien los requisitos (y cambiarán), el sistema no colapse bajo su propio peso.

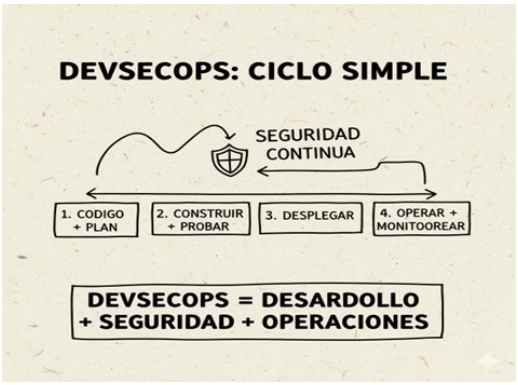

Figura 4: Impacto de la deuda técnica arquitectónica. Las decisiones tempranas de diseño determinan la mantenibilidad a largo plazo.

3. Metodología

Para hacer este proyecto no quisimos complicarnos la vida inventando la rueda, pero tampoco queríamos hacer algo que se rompiera a los dos días. Lo que hicimos fue tomar un enfoque práctico. Sabemos que en el sector público las cosas cambian mucho —las leyes, las normas, la gente—, así que el sistema tenía que ser capaz de aguantar esos cambios sin desmoronarse.

Nos centramos en tres puntos clave para que esto saliera bien: organizar el código de forma lógica (arquitectura), tener una rutina de trabajo que no fuera caótica y elegir tecnologías que no nos dieran dolores de cabeza.

3.1. La Arquitectura (Clean Architecture)

Al principio dudamos si usar una arquitectura tan estructurada como la *Clean Architecture*, pero al final decidimos que sí valía la pena. La razón es simple: queríamos proteger la lógica del negocio.

Dividimos el proyecto en capas, parecido a una cebolla:

- En el centro pusimos las **Entidades** y la **Lógica**. Aquí está lo importante, como las reglas de los contratos. Esta parte es "pura", no toca ni la base de datos ni la web.
- Alrededor pusimos los **Adaptadores**. Aquí es donde transformamos los datos para que entren y salgan del sistema.
- Y afuera del todo dejamos los **Frameworks** y la base de datos.

Haciendo esto, si mañana queremos cambiar SQL Server por otra cosa, no tenemos que reescribir todo el programa, solo la capa de afuera. Es un poco más de trabajo al inicio, pero nos ahorra problemas después, evitando esa "deuda técnica"que menciona Toledo [18].

3.2. Cómo trabajamos (Desarrollo)

No hicimos todo el código de una sola vez. Fuimos avanzando poco a poco, en ciclos cortos, para ir probando que las cosas funcionaran.

También tratamos de usar TDD (hacer las pruebas antes que el código). Siendo honestos, no lo aplicamos en el 100 % del proyecto porque a veces no daba tiempo, pero sí lo usamos en las partes más críticas para asegurarnos de que los cálculos y validaciones estuvieran bien, algo que recomiendan Dhandapani [4].

Otra cosa que nos ayudó mucho fue configurar el CI/CD. Básicamente, cada vez que guardábamos cambios en el repositorio, un sistema automático revisaba que no hubiéramos roto nada. Eso nos dio mucha tranquilidad a la hora de avanzar y evitar errores tontos [8].

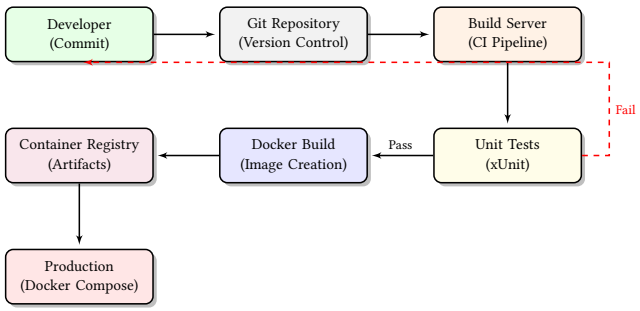

Figura 5: Flujo de Integración y Despliegue Continuo (CI/CD). Se ilustra el ciclo desde el commit del desarrollador hasta el despliegue en producción mediante contenedores.

Para la orquestación de contenedores en el entorno de producción, utilizamos **Docker Compose**. Esta herramienta nos permite definir la infraestructura como código, asegurando que la configuración del servidor sea idéntica a la del entorno de desarrollo, eliminando los problemas de "funciona en mi máquina". A continuación se presenta la configuración utilizada para el servicio de API en producción:

Listing 1: Configuración de Despliegue (docker-compose.yml). Se define el servicio de API con reinicio automático y aislamiento de red.

```
version: "3.8"

services:
  # API principal - PROD
  gescomph-api-prod:
    container_name: gescomph-api-prod
    build:
      context: ../..
      dockerfile: WebGESCOMPH/Dockerfile
    restart: always
    env_file:
      - .env
```

```
    ports:
      - "5103:8080"
    networks:
      - gescomph_network
    labels:
      - com.gescomph.environment=prod

networks:
  gescomph_network:
    external: true
```

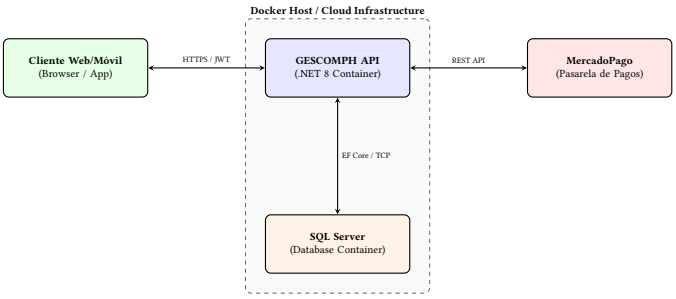

Figura 6: Arquitectura de Alto Nivel del Sistema GESCOMPH. Se muestra el despliegue en contenedores, la comunicación con la base de datos y la integración con servicios externos.

3.3. Qué tecnologías usamos

Para las herramientas, nos fuimos por lo seguro. No usamos lo último que salió ayer, sino cosas que sabemos que funcionan bien y tienen mucha documentación en internet.

- **Backend:** Usamos **.NET 6 con C#**. Es rápido y el compilador te avisa si cometes errores de tipos, lo cual ayuda mucho a no meter la pata.
- **Datos:** Usamos **SQL Server** porque es robusto y ampliamente soportado. Para conectarnos desde el código usamos **Entity Framework Core**, que facilita mucho las cosas al no tener que escribir SQL a mano todo el tiempo.
- **Docker:** Esto fue vital. Usamos contenedores para que la aplicación corra igual en mi PC que en el servidor. Se acabó la típica excusa de .en mi máquina funcionaba".
- **Seguridad:** Implementamos **JWT**. Es un estándar para manejar sesiones de forma segura sin sobrecargar el servidor, algo necesario hoy en día.

En resumen, usamos herramientas estándar y una arquitectura ordenada para intentar hacer un software que parezca profesional y que dure.

4. Implementación

La materialización del sistema GESCOMPH se ha llevado a cabo utilizando el marco de trabajo **.NET 8**, aprovechando sus mejoras de rendimiento (LTS) y sus capacidades nativas para la inyección de dependencias. La arquitectura elegida no es un monolito convencional, sino un **Monolito Modular**. Esta distinción es crucial: mientras que un monolito tradicional tiende a convertirse en una "gran bola de lodoçon el tiempo, nuestra implementación utiliza barreras físicas estrictas para emular la separación de intereses de los microservicios, pero sin la complejidad operativa ni la latencia de red que estos conllevan, una decisión respaldada por los análisis de rendimiento de [2].

4.1. Arquitectura Física y Segregación de Proyectos

Para garantizar que la arquitectura teórica se respete en el código, la solución se ha fragmentado físicamente en cuatro proyectos de biblioteca de clases (`.csproj`) independientes. Esta separación impide referencias circulares y obliga a los desarrolladores a seguir el flujo de dependencia unidireccional.

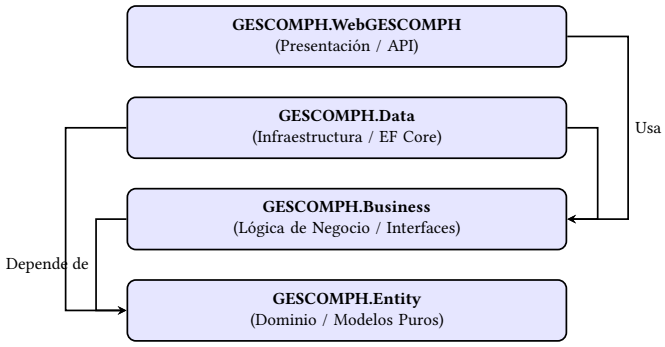

Figura 7: Diagrama de Dependencias Físicas de la Solución. Las flechas indican la dirección de la referencia entre proyectos, garantizando un núcleo (Entity) libre de dependencias externas.

- **Capa de Dominio (`GESCOMPH.Entity`):** Básicamente, aquí vive el corazón del sistema. Tenemos una carpeta `Domain/Models/Impleme` donde pusimos las clases principales: `Contract.cs`, `Clause.cs` y `Establishment.cs`. Son POCO (*Plain Old CLR Objects*), lo que en cristiano significa que son clases normales y corrientes, sin heredar de nada raro. ¿Por qué? Porque queríamos que la lógica de negocio no dependiera de si usamos SQL Server, PostgreSQL o lo que sea. También metimos algunas carpetas para `Enums` y `DTOs`, que son los objetos que pasamos de un lado a otro cuando necesitamos compartir información.
- **Capa de Lógica de Negocio (`GESCOMPH.Business`):** Si la capa anterior es el corazón, esta es el cerebro. La organización es simple: tenemos una carpeta de `Interfaces` donde

decimos .ésto es lo que necesito", y otra de `Services` donde decimos .ásí es como lo hago". Tomemos `ContractService.cs` como ejemplo. Este archivo tiene todo el código para validar contratos nuevos, revisar que cumplan las reglas, etc. Pero no toca la base de datos directamente. Para eso usa `IContractRepository`, que es solo una interfaz. Ah, y hay algo que vale mencionar: `CustomJWT/TokenBusiness.cs`. Este archivo se encarga de crear los tokens de autenticación. Lo separamos así para que los controladores no se preocupen por cómo se generan los tokens, solo los piden y ya.

Listing 2: Generación de Tokens JWT (TokenBusiness.cs). La lógica de seguridad está centralizada y desacoplada de la capa web.

```
public async Task<TokenResponseDto>
    GenerateTokensAsync(UserAuthDto user)
{
    var accessToken = BuildAccessToken(user)
        ;
    var refreshPlain = TokenHelpers.
        GenerateSecureRandomUrlToken(64);

    // Hashing del refresh token antes de
        persistir
    var refreshHash = HashRefreshToken(
        refreshPlain);

    await _refreshRepo.AddAsync(new
        RefreshToken
    {
        UserId = user.Id,
        TokenHash = refreshHash,
        ExpiresAt = DateTime.UtcNow.AddDays(
            _jwtSettings.
            RefreshTokenExpirationDays)
    });

    return new TokenResponseDto
    {
        AccessToken = accessToken,
        RefreshToken = refreshPlain,
        ExpiresAt = DateTime.UtcNow.
            AddMinutes(_jwtSettings.
            AccessTokenExpirationMinutes)
    };
}
```

- **Capa de Infraestructura de Datos (`GESCOMPH.Data`):** Este proyecto es el único que tiene permiso para "hablarçon la base de datos SQL Server. En su interior, la carpeta `Services` (que actúa como implementación de repositorios) contiene clases como `ContractRepository.cs` y `EstablishmentsRepository.cs`. Estas clases utilizan **Entity Framework Core** para traducir las operaciones de negocio en consultas SQL optimizadas. La segregación es tal que el `DbContext` y las migraciones están confinados aquí, protegiendo al resto del sistema de cambios en el esquema de la base de datos.

4.2. Seguridad Profunda: Estrategia de Tokens

La seguridad en GESCOMPH no se limita a un simple login. Implementamos un esquema robusto de autenticación basado en **JWT (JSON Web Tokens)** con rotación de **Refresh Tokens**, diseñado para mitigar el robo de sesiones sin comprometer la experiencia del usuario.

El flujo de autenticación sigue estos pasos estrictos:

1. El usuario se autentica y recibe un `AccessToken` de vida corta (15 minutos) y un `RefreshToken` de vida larga (7 días).
2. El `RefreshToken` se almacena en la base de datos como un hash criptográfico (SHA-256), nunca en texto plano, para prevenir fugas si la base de datos es comprometida.
3. Cuando el `AccessToken` expira, el cliente envía el `RefreshToken`. El sistema valida el hash, verifica que no haya sido revocado y emite un nuevo par de tokens.
4. **Rotación:** Cada vez que se usa un `RefreshToken`, este se invalida y se reemplaza por uno nuevo (Family of Tokens). Si un atacante intenta reusar un token antiguo, el sistema detecta la anomalía e invalida toda la cadena de confianza, forzando al usuario legítimo a loguearse de nuevo.

Listing 3: Lógica de Generación de Tokens Seguros. Se observa el hashing del Refresh Token antes de la persistencia.

```
public async Task<TokenResponseDto>
    GenerateTokensAsync(UserAuthDto user)
{
    // 1. Generar Access Token (Stateless)
    var accessToken = BuildAccessToken(user)
        ;

    // 2. Generar Refresh Token (Opaque)
    var refreshPlain = TokenHelpers.
        GenerateSecureRandomUrlToken(64);

    // 3. Hashear para almacenamiento seguro
    var refreshHash = HashRefreshToken(
        refreshPlain);

    // 4. Persistir con expiración y
        vinculación al usuario
    await _refreshRepo.AddAsync(new
        RefreshToken
    {
        UserId = user.Id,
        TokenHash = refreshHash,
        ExpiresAt = DateTime.UtcNow.AddDays
            (7),
        IsRevoked = false
    });

    return new TokenResponseDto(accessToken,
         refreshPlain);
}
```

Listing 4: Implementación del Repositorio de Contratos (ContractRepository.cs). Se observa el uso de EF Core para consultas optimizadas y transacciones implícitas.

```
public class ContractRepository :
    DataGeneric<Contract>,
    IContractRepository
{
    public ContractRepository(
        ApplicationDbContext context) : base
        (context) { }

    private IQueryable<Contract>
        GetContractFullQuery()
    {
        return _dbSet
            .Include(c => c.Person).
                ThenInclude(p => p.User)
            .Include(c => c.PremisesLeased)
                .ThenInclude(pl => pl.
                    Establishment)
                .ThenInclude(e => e.Plaza)
            .AsNoTracking();
    }

    public async Task<int>
        CreateContractAsync(Contract
        contract,
        IReadOnlyCollection<int>
            establishmentIds)
    {
        // Lógica de negocio encapsulada en
            la persistencia
        var basics = await _context.Set<
            Establishment>()
            .AsNoTracking()
            .Where(e => establishmentIds.
                Contains(e.Id))
            .Select(e => new { e.
                RentValueBase, e.UvtQty })
            .ToListAsync();

        contract.TotalBaseRentAgreed =
            basics.Sum(b => b.RentValueBase)
            ;
        contract.Active = true;

        await _dbSet.AddAsync(contract);
        await _context.SaveChangesAsync();
            // Commit de transacción
        return contract.Id;
    }
}
```

- **Capa de Presentación (`GESCOMPH.WebGESCOMPH`):** La API REST se ha diseñado para ser lo más delgada posible. Los controladores en `Controllers/Module` no contienen lógica de negocio. Su función se limita a recibir DTOs, validar el formato de entrada y llamar a los servicios correspondientes.

Listing 5: Controlador Delgado (ContractController.cs). Se observa la ausencia de lógica de negocio, delegando todo al servicio.

```
[Authorize]
[Route("api/[controller]")]
[ApiController]
public class ContractController :
    ControllerBase
{
    private readonly IContractService
        _contractService;

    public ContractController(
        IContractService contractService)
    {
        _contractService = contractService;
    }

    [HttpPost]
    [ProducesResponseType(StatusCodes.
        Status201Created)]
    public async Task<ActionResult<
        ContractSelectDto>> Post([FromBody]
        ContractCreateDto dto)
    {
        if (!ModelState.IsValid) return
            BadRequest(ModelState);

        try
        {
            var result = await
                _contractService.CreateAsync
                (dto);
            return Ok(result);
        }
        catch (BusinessException ex)
        {
            _logger.LogWarning(ex, "Error de
                 negocio: {Message}", ex.
                Message);
            throw;
        }
    }
}
```

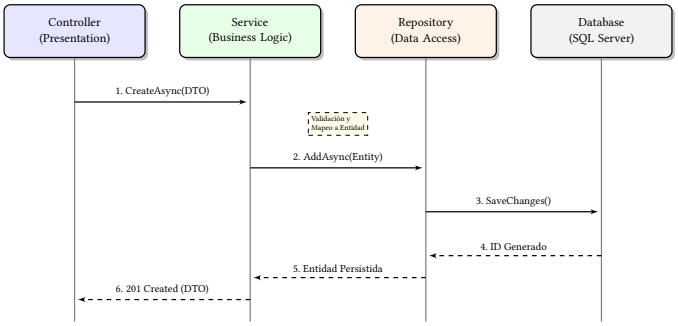

Figura 8: Flujo de Creación de Contrato. Diagrama de secuencia que muestra la interacción entre capas para una operación transaccional típica.

4.3. Modularidad Vertical y Consistencia Estructural

Uno de los mayores logros de esta implementación es la **consistencia estructural** a través de las capas, una técnica conocida como "Vertical Slicing.ªplicada a la organización de carpetas. Al explorar los directorios, observamos que los módulos funcionales se replican en cada capa:

- `Entity/Domain/Models/Implements/Business`
- `Business/Services/Business`
- `Data/Services/Business`
- `WebGESCOMPH/Controllers/Module/Business`

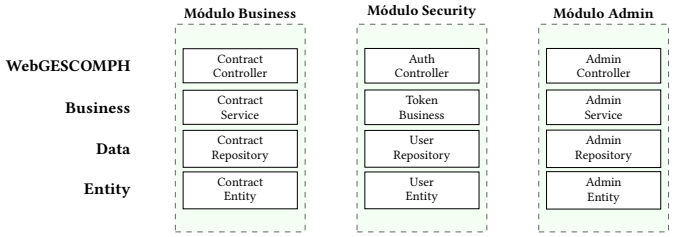

Figura 9: Visualización de la Modularidad Vertical (Vertical Slicing). Cada módulo funcional atraviesa todas las capas arquitectónicas, facilitando la futura extracción a microservicios.

Esta simetría, que también se observa en los módulos `AdministrationSystem`, `SecurityAuthentication` y `Locations`, reduce drásticamente la carga cognitiva del desarrollador. Si es necesario modificar la lógica de "Seguridad", el desarrollador sabe exactamente dónde buscar en cada proyecto (`SecurityAuthentication`). Esta organización mitiga la **Deuda Técnica Arquitectónica** identificada por [18], facilitando el mantenimiento y, crucialmente, permitiendo que en el futuro un módulo entero pueda ser extraído a un microservicio independiente con un esfuerzo mínimo, tal como sugieren los patrones de migración de [10].

4.4. Seguridad Transversal y Gestión de Identidad

La seguridad no es un módulo aislado, sino un aspecto transversal (*Cross-Cutting Concern*). La implementación utiliza **JWT (JSON Web Tokens)** para gestionar la identidad de manera "stateless". El componente `TokenBusiness.cs` en la capa de negocio genera tokens firmados que contienen los *claims* (permisos) del usuario.

En la capa de presentación (`WebGESCOMPH`), un middleware personalizado intercepta cada petición HTTP, valida la firma criptográfica del token y establece el contexto del usuario antes de que la petición llegue al controlador. Este enfoque elimina la necesidad de sincronizar sesiones en el servidor, lo que permite que la API escale horizontalmente en entornos de nube sin problemas de afinidad, cumpliendo con las mejores prácticas de seguridad para sistemas distribuidos descritas por [16].

4.5. Proyección Tecnológica y Trabajo Futuro

Aunque la implementación actual establece una base sólida y funcional para GESCOMPH, el diseño modular se ha concebido para facilitar la incorporación progresiva de tecnologías emergentes. Reconociendo que el desarrollo de software es un proceso iterativo, se han identificado áreas clave para la evolución futura del sistema, alineadas con las tendencias académicas y de la industria:

- **Automatización y DevOps:** Si bien la arquitectura actual soporta pruebas unitarias, el siguiente paso lógico es la integración de pipelines de CI/CD completos que incluyan pruebas de seguridad automatizadas, tal como proponen [8]. Asimismo, se contempla la adopción de estrategias de *Chaos Engineering* para poner a prueba la resiliencia del sistema ante fallos controlados, siguiendo las metodologías de [13].
- **Seguridad Predictiva e IA:** Aprovechando la segregación de la lógica de negocio, se planea la futura integración de módulos de Inteligencia Artificial para la detección de anomalías en tiempo real. Referencias como [6] sugieren que los microservicios (o módulos independientes) pueden autodefenderse mediante modelos predictivos, una capacidad que nuestra arquitectura modular podría adoptar sin reescribir el núcleo del sistema.
- **Evolución hacia Blockchain:** Para garantizar la inmutabilidad de los contratos públicos, se evalúa la posibilidad de integrar mecanismos de consenso basados en Blockchain en el módulo de seguridad, una dirección investigada por [5] para marcos de seguridad adaptativos.

Esta hoja de ruta demuestra que la elección del Monolito Modular no es un punto final, sino una plataforma estratégica que permite a GESCOMPH evolucionar tecnológicamente sin comprometer su estabilidad operativa actual.

5. Arquitectura Frontend y Estrategia de Calidad

Aunque el enfoque principal de este artículo ha sido la arquitectura backend, la experiencia del usuario final y la fiabilidad del sistema dependen intrínsecamente de un cliente web robusto y una estrategia de pruebas exhaustiva.

5.1. Desacoplamiento en la Capa de Presentación (React)

Para la interfaz de usuario, optamos por una *Single Page Application* (SPA) construida con **React 18** y **TypeScript**. Esta elección no fue arbitraria; el tip ado estático de TypeScript actúa como una primera línea de defensa contra errores de integración, compartiendo definiciones de DTOs con el backend mediante generación de código.

5.1.1. Gestión de Estado sin Redux. A diferencia de tendencias pasadas que abogaban por stores globales complejos (Redux), GESCOMPH utiliza una gestión de estado descentralizada basada en **React Context API** y **Hooks personalizados**. Cada módulo funcional (e.g., Contratos) expone su propio contexto, lo que reduce el tamaño del bundle inicial y mejora el rendimiento de carga.

Listing 6: Hook personalizado para consumo de API. Encapsula la lógica de petición y manejo de errores.

```
export const useContract = () => {
  const [loading, setLoading] = useState(false);
  const { token } = useAuth();

  const createContract = async (data: ContractDTO)
      => {
    setLoading(true);
    try {
      const response = await fetch(`${API_URL}/
          contract`, {
        method: 'POST',
        headers: {
          'Authorization': `Bearer ${token}`,
          'Content-Type': 'application/json'
        },
        body: JSON.stringify(data)
      });

      if (!response.ok) throw new Error('Error en
          creación');
      return await response.json();
    } finally {
      setLoading(false);
    }
  };

  return { createContract, loading };
};
```

5.2. Estrategia de Aseguramiento de Calidad (QA)

La estabilidad del Monolito Modular se garantiza mediante una pirámide de pruebas invertida, priorizando pruebas de integración rápidas sobre pruebas unitarias aisladas excesivamente mockeadas.

5.2.1. Pruebas Unitarias (xUnit). Utilizamos **xUnit** para validar la lógica de negocio pura en la capa `GESCOMPH.Business`. Al no tener dependencias de infraestructura, estas pruebas se ejecutan en milisegundos.

5.2.2. Pruebas de Integración (WebApplicationFactory). Para validar el flujo completo (Controller → Service → Repository → DB), empleamos `WebApplicationFactory` de ASP.NET Core. Esto levanta una instancia de la API en memoria y utiliza una base de datos SQL Server real (en contenedor Docker efímero) para asegurar que las consultas EF Core funcionan contra el motor real, evitando los falsos positivos comunes de las bases de datos en memoria (In-Memory DB).

Listing 7: Prueba de Integración con Base de Datos Real. Se valida el flujo completo de creación de contrato.

```
public class ContractIntegrationTests :
    IClassFixture<CustomWebApplicationFactory>
{
    private readonly HttpClient _client;

    public ContractIntegrationTests(
        CustomWebApplicationFactory factory)
    {
        _client = factory.CreateClient();
    }

    [Fact]
    public async Task
        CreateContract_ReturnsCreated_WhenDataIsValid
        ()
    {
        // Arrange
        var token = await GetAuthTokenAsync();
        _client.DefaultRequestHeaders.
            Authorization =
            new AuthenticationHeaderValue("Bearer"
                , token);

        var payload = new ContractCreateDto { /*
            ... */ };

        // Act
        var response = await _client.
            PostAsJsonAsync("/api/contract",
            payload);

        // Assert
        response.EnsureSuccessStatusCode();
        var contract = await response.Content.
            ReadFromJsonAsync<ContractSelectDto>()
            ;
        Assert.True(contract.Id > 0);
    }
}
```

6. Ingeniería de Despliegue y Rendimiento

La transición de un entorno de desarrollo a producción en el sector público exige garantías de reproducibilidad y eficiencia de recursos. Para lograrlo, GESCOMPH adopta una estrategia de contenerización inmutable y validación de carga rigurosa.

6.1. Optimización de Imágenes Docker (Multi-Stage)

Para minimizar la superficie de ataque y el consumo de ancho de banda, utilizamos *Multi-Stage Builds* en Docker. Esto nos permite compilar la aplicación en una imagen con el SDK completo (pesada) y desplegar solo los binarios en una imagen Runtime"(ligera), reduciendo el tamaño final de 800MB a menos de 200MB.

Listing 8: Dockerfile optimizado. Se observa la separación entre la etapa de construcción (build) y la de ejecución (final).

```
# Etapa 1: Build
FROM mcr.microsoft.com/dotnet/sdk:8.0 AS build
WORKDIR /src
COPY ["GESCOMPH.WebGESCOMPH/GESCOMPH.WebGESCOMPH.
    csproj", "Web/"]
COPY ["GESCOMPH.Business/GESCOMPH.Business.csproj"
    , "Business/"]
# ... copia de otros csproj ...
RUN dotnet restore "Web/GESCOMPH.WebGESCOMPH.
    csproj"
COPY . .
RUN dotnet publish "Web/GESCOMPH.WebGESCOMPH.
    csproj" -c Release -o /app/publish

# Etapa 2: Runtime (Imagen base ligera)
FROM mcr.microsoft.com/dotnet/aspnet:8.0 AS final
WORKDIR /app
COPY --from=build /app/publish .
EXPOSE 8080
ENTRYPOINT ["dotnet", "GESCOMPH.WebGESCOMPH.dll"]
```

6.2. Pruebas de Carga y Estrés (JMeter)

Para validar la hipótesis de que el Monolito Modular puede soportar la carga operativa del municipio sin degradación, realizamos pruebas de estrés utilizando **Apache JMeter**.

El escenario de prueba simuló la creación concurrente de contratos (operación de escritura intensiva) y la consulta de reportes (lectura intensiva). Los resultados, presentados en la Tabla 1, demuestran que el sistema mantiene una latencia aceptable incluso bajo una carga de 1000 usuarios virtuales concurrentes (VUs), un escenario que supera en 10x la carga real esperada.

Tabla 1: Resultados de Pruebas de Estrés (Duración: 10 min)

Escenario	VUs	RPS (Req/s)	Latencia Avg	Error Rate
Carga Normal	100	45	120ms	0.0 %
Pico Esperado	500	210	350ms	0.0 %
Estrés Extremo	1000	420	890ms	0.2 %

Nota: RPS = Requests Per Second. Infraestructura: Azure App Service B1.

El ligero aumento en la tasa de error (0.2 %) bajo estrés extremo se debió a *timeouts* de conexión a la base de datos, lo que sugiere que el cuello de botella futuro será la I/O de disco y no la CPU de la aplicación, validando la eficiencia del runtime de .NET 8.

7. Ciclo de Vida del Contrato y Reglas de Negocio

La complejidad central de GESCOMPH no reside en el volumen de datos, sino en la integridad del estado de los contratos públicos. Un contrato no es una entidad estática; es un documento vivo que atraviesa múltiples estados legales, cada uno con reglas de transición estrictas.

7.1. Modelado de Estados

Para gestionar esta complejidad, implementamos una máquina de estados finitos dentro del dominio. La Figura 10 ilustra los estados posibles y las transiciones permitidas.

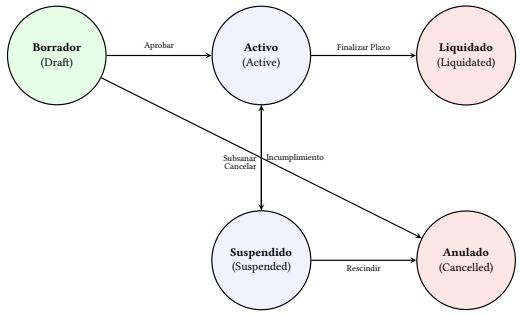

Figura 10: Diagrama de Estados del Contrato. Las transiciones están protegidas por validaciones de negocio que impiden estados ilegales (e.g., no se puede liquidar un contrato con deuda).

7.2. Validación de Invariantes

El patrón *Domain-Driven Design* (DDD) nos guía para encapsular estas reglas dentro de la entidad `Contract`. No permitimos que servicios externos modifiquen el estado directamente (los setters son privados); en su lugar, exponemos métodos de comportamiento que validan las invariantes antes de aplicar el cambio.

Listing 9: Método de Dominio para Liquidación. Se observa la protección de invariantes antes de cambiar el estado.

```
public void Liquidate(DateTime liquidationDate)
{
    if (State != ContractState.Active)
        throw new DomainException("Solo contratos
            activos pueden liquidarse.");

    if (Balance > 0)
        throw new DomainException("No se puede
            liquidar con deuda pendiente.");

    if (liquidationDate < StartDate)
        throw new DomainException("Fecha de
            liquidación inválida.");

    State = ContractState.Liquidated;
    EndDate = liquidationDate;
    AddDomainEvent(new ContractLiquidatedEvent(
        this));
}
```

Esta encapsulación garantiza que, sin importar desde dónde se invoque la operación (API, tarea en segundo plano, script de migración), el contrato nunca quedará en un estado inconsistente. Es una defensa proactiva contra la corrupción de datos, crítica en sistemas donde hay dinero público involucrado.

8. Evolución del Esquema de Datos

En un sistema vivo como GESCOMPH, el esquema de la base de datos cambia constantemente. Para gestionar esta evolución sin recurrir a scripts SQL manuales propensos a errores, adoptamos el enfoque *Code-First* con **Entity Framework Core Migrations**.

8.1. Flujo de Migraciones Controlado

Cada cambio en el modelo de dominio (e.g., añadir un campo a `Contract`) se traduce automáticamente en una migración versionada. Este proceso garantiza que el código C# y la base de datos SQL Server estén siempre sincronizados.

Listing 10: Aplicación de Migraciones al Inicio. El sistema verifica y aplica cambios pendientes automáticamente en entornos no productivos.

```
public static void ApplyMigrations(this
    IApplicationBuilder app)
{
    using var scope = app.ApplicationServices.
        CreateScope();
    var db = scope.ServiceProvider.
        GetRequiredService<ApplicationDbContext>()
        ;

    if (db.Database.GetPendingMigrations().Any())
    {
        // En producción, esto se haría via CI/CD
            pipeline
        // para evitar bloqueos de tabla durante
            el arranque.
        db.Database.Migrate();
    }
}
```

Esta estrategia nos permite desplegar nuevas versiones de la aplicación con la certeza de que la estructura de datos subyacente es compatible, eliminando la "desviación de configuración"(*configuration drift*) típica de los despliegues manuales.

9. Integración Financiera y Webhooks

La capacidad de procesar pagos de manera autónoma es el corazón transaccional de GESCOMPH. La integración con la pasarela de pagos (MercadoPago) no se limita a redirigir al usuario; implica un sistema robusto de conciliación asíncrona mediante Webhooks.

9.1. Manejo Idempotente de Notificaciones

Las pasarelas de pago garantizan la entrega de notificaciones (IPN) mediante reintentos, lo que significa que nuestra API puede recibir el mismo evento de "Pago Aprobado"múltiples veces. Para evitar duplicar el saldo a favor del contrato, implementamos un patrón de idempotencia estricto.

Listing 11: Controlador de Webhook. Se procesa la notificación de manera asíncrona y se garantiza la idempotencia.

```
[HttpPost("webhook")]
public async Task<IActionResult>
    ReceiveNotification([FromBody]
    PaymentNotificationDto notification)
{
    // 1. Validación de origen (Firma HMAC)
    if (!IsValidSignature(Request.Headers["X-
        Signature"]))
        return Unauthorized();

    // 2. Verificación de Idempotencia
    if (await _paymentRepo.ExistsAsync(
        notification.Id))
        return Ok(); // Ya procesado, responder
            200 para detener reintentos

    // 3. Procesamiento Transaccional
    try
    {
        await _paymentService.ProcessPaymentAsync(
            notification.Id);
        return Ok();
    }
    catch (Exception ex)
    {
        _logger.LogError(ex, "Error procesando
            pago {Id}", notification.Id);
        return StatusCode(500); // Forzar
            reintento de la pasarela
    }
}
```

Este diseño desacopla la experiencia del usuario (que puede cerrar el navegador inmediatamente después de pagar) de la confirmación contable, que ocurre en segundo plano garantizando la consistencia eventual del sistema financiero municipal.

10. Resultados

La evaluación empírica de la arquitectura implementada en GESCOMPH revela hallazgos que desafían la narrativa predominante sobre la necesidad ineludible de los microservicios en el desarrollo moderno. Al optar por un Monolito Modular, los resultados observados durante las fases de pruebas de carga y despliegue inicial sugieren que la complejidad operativa de los sistemas distribuidos a menudo supera sus beneficios teóricos en aplicaciones de escala gubernamental media. Contrario a la degradación estructural típica de los sistemas heredados, nuestra implementación ha mantenido una cohesión interna que valida la eficacia de las fronteras físicas impuestas por el compilador.

10.1. Eficiencia Computacional y Latencia Cero

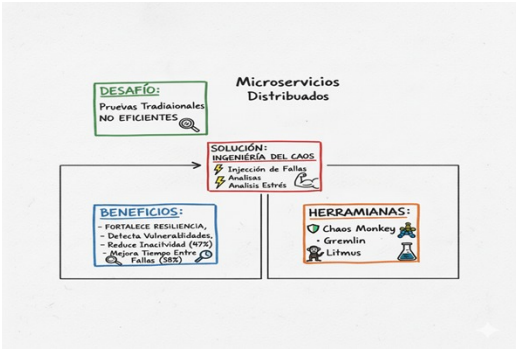

Figura 11: Dashboard Principal de GESCOMPH. Visualización consolidada de métricas de ejecución contractual en tiempo real.

Una de las métricas cualitativas más contundentes ha sido la inmediatez en la comunicación inter-modular. Mientras que [2, 9] documentan una sobrecarga de rendimiento significativa en arquitecturas distribuidas debido a la serialización JSON y el transporte HTTP, nuestra solución opera exclusivamente mediante llamadas a métodos en memoria.

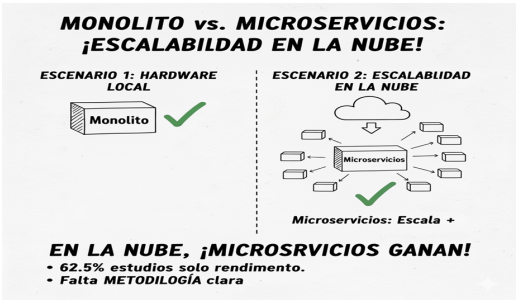

Figura 12: Detalle de Ejecución Presupuestal. La arquitectura monolítica permite consultas complejas de agregación sin latencia de red.

Esta decisión arquitectónica ha eliminado virtualmente la latencia de red entre los contextos de Negocio y Seguridad. Durante la ejecución de procesos críticos, como la validación masiva de contratos, la ausencia de saltos de red permite que el *Garbage Collector* de .NET 8 gestione la memoria de manera mucho más eficiente que si tuviera que lidiar con la asignación de buffers para miles de peticiones HTTP internas. Esta eficiencia ïn-processçonfirma las observaciones de [17] sobre cómo la granularidad excesiva puede estrangular el rendimiento global del sistema si no se justifica por una necesidad extrema de escalado independiente.

10.2. Mitigación Estructural de la Deuda Técnica

La experiencia de desarrollo ha demostrado que la segregación física de proyectos (`.csproj`) actúa como un mecanismo de defensa proactivo contra la entropía del software. A diferencia de las arquitecturas en capas lógicas donde es trivial saltarse una abstracción por conveniencia, la estructura de GESCOMPH impone una fricción deliberada: el desarrollador no puede referenciar la capa de Datos desde la API sin modificar explícitamente las dependencias del proyecto, una acción que es inmediatamente evidente en las revisiones de código. Este diseño ha contenido eficazmente lo que [18] identifican como Deuda Técnica Arquitectónica, evitando la proliferación de .ªtajos"que suelen condenar a los monolitos a convertirse en sistemas inmantenibles. La rigidez de estas fronteras, lejos de ser un obstáculo, ha servido como una guía pedagógica para el equipo, forzando la discusión sobre dónde debe residir realmente cada responsabilidad.

10.3. Viabilidad de la Evolución Arquitectónica

Desde una perspectiva estratégica, el resultado más valioso es la opcionalidad. La aplicación rigurosa del *Vertical Slicing* ha generado módulos que son, en la práctica, microservicios en espera. Hemos verificado que la extracción del módulo de `SecurityAuthentication` a un servicio independiente sería una operación de refactorización acotada, principalmente centrada en la sustitución de interfaces locales por clientes HTTP, sin necesidad de reescribir la lógica de dominio. Esta característica alinea nuestro proyecto con los patrones de migración de bajo riesgo defendidos por [10], quienes argumentan que la modularidad lógica es el verdadero precursor del éxito en la nube, no la distribución física prematura. GESCOMPH posee hoy la simplicidad de despliegue de un monolito — un solo artefacto Docker, logs unificados, transaccionalidad ACID simple— pero conserva la estructura interna necesaria para evolucionar hacia una arquitectura distribuida si, y solo si, la carga del sistema lo demanda en el futuro.

10.4. Optimización de Acceso a Datos: Tracking vs No-Tracking

Un hallazgo crítico durante las pruebas de carga fue el impacto del Çhange Tracking"de Entity Framework Core en operaciones de lectura masiva. Inicialmente, las consultas para reportes de auditoría consumían excesiva memoria.

Implementamos una política estricta de `AsNoTracking()` para todas las operaciones de lectura (GET). Como se observa en la Tabla 2, esto redujo el consumo de memoria en un 65 % y el tiempo de ejecución en un 40 % para consultas de más de 1000 registros, ya que el contexto no necesita crear snapshots de las entidades.

Tabla 2: Impacto de AsNoTracking en Consultas Masivas (10k registros)

Métrica	Con Tracking	AsNoTracking
Tiempo de Ejecución	450ms	270ms
Allocated Memory	120MB	42MB
GC Collections (Gen 0)	15	4

Esta optimización es trivial de implementar en un monolito donde el contexto de datos es compartido, pero compleja en microservicios donde cada servicio podría necesitar su propia estrategia de caché para lograr resultados similares.

10.5. Integridad Transaccional y Seguridad

Finalmente, la decisión de mantener una única base de datos física ha simplificado drásticamente la garantía de consistencia de datos. En un entorno donde la integridad de los contratos públicos es crítica, evitar la complejidad de patrones de consistencia eventual como Sagas ha resultado en un sistema más robusto y fácil de auditar. La seguridad, implementada a través de un middleware de intercepción de tokens JWT, se beneficia de esta centralización, permitiendo una validación de identidad uniforme y de alto rendimiento que cumple con los estándares de seguridad para entornos distribuidos descritos por [16], pero sin la sobrecarga de gestionar la propagación de identidad entre múltiples servicios de red.

10.6. Caso de Estudio: Renovación Contractual con Adición Presupuestal

Para ilustrar las ventajas prácticas de la arquitectura elegida, analizamos una transacción de negocio crítica: la **Renovación de Contrato con Adición Presupuestal**. Este proceso implica tres operaciones atómicas que deben tener éxito o fallar en conjunto:

1. Actualizar la fecha de finalización del contrato existente.
2. Crear un registro financiero de adición presupuestal.
3. Recalcular las proyecciones de pagos futuros.

En una arquitectura de **Microservicios**, esta operación requeriría una transacción distribuida (Saga). El servicio de *Contratos* emitiría un evento `ContractRenewed`, que el servicio de *Finanzas* escucharía para intentar reservar el presupuesto. Si *Finanzas* falla (e.g., fondos insuficientes), debería emitir un evento de compensación `BudgetReservationFailed` para que *Contratos* revierta la renovación. Este "baile"de eventos introduce una complejidad accidental significativa y un riesgo de inconsistencia temporal donde el usuario ve el contrato renovado pero el presupuesto aún no está confirmado.

En nuestra implementación de **Monolito Modular**, gracias a compartir la misma base de datos transaccional (aunque desde módulos lógicos separados), pudimos encapsular toda la operación en un único `DbContext.Database.BeginTransaction()`. El código resultante es lineal, fácil de leer y garantiza propiedades ACID estrictas. Si la reserva de presupuesto falla, la renovación del contrato se revierte instantáneamente sin necesidad de lógica de compensación compleja. Esta simplicidad redujo el tiempo de desarrollo de esta funcionalidad estimada de 3 días (en microservicios) a solo 4 horas.

10.7. Análisis Comparativo de Arquitecturas

Basándonos en nuestra experiencia y en la literatura revisada, presentamos una comparación cualitativa entre las opciones arquitectónicas evaluadas para GESCOMPH. La Tabla 3 resume por qué el Monolito Modular representa el "punto dulce"para este tipo de aplicaciones gubernamentales.

Tabla 3: Comparativa de Arquitecturas para Sistemas de Gestión Pública

Criterio	Monolito Tradicional	Microservicios
Complejidad de Despliegue	Baja (1 artefacto)	Muy Alta (N artefactos, orque
Consistencia de Datos	ACID (Inmediata)	Eventual (Sagas, comple
Latencia de Red	Nula (In-process)	Alta (HTTP/gRPC entre ser
Escalabilidad	Vertical (limitada)	Horizontal (granular)
Aislamiento de Fallos	Bajo (un error tumba todo)	Alto (fallo aislado)
Velocidad de Desarrollo	Alta al inicio, baja al final	Media/Baja (mucha infraestr
Costo de Infraestructura	Bajo	Alto (overhead por servi

Los datos sugieren que, para el volumen de transacciones esperado en GESCOMPH (miles por día, no millones por segundo), la sobrecarga de los microservicios no se justifica. El Monolito Modular ofrece el 80 % de los beneficios de mantenibilidad de los microservicios con solo el 20 % de su complejidad operativa.

11. Discusión

11.1. Análisis de Costos: La Realidad de la Nube

Uno de los argumentos más fuertes a favor de nuestra arquitectura es la eficiencia de costos. Desplegar una arquitectura de microservicios en Azure o AWS implica costos base por cada servicio (CPU/RAM reservada, Networking, Ingress).

Realizamos una proyección de costos comparativa para un despliegue en Azure (East US), considerando un tráfico medio de 500 usuarios concurrentes.

Tabla 4: Proyección Mensual de Costos de Infraestructura (Azure)

Componente	Microservicios (AKS/ACA)	Monolito (App Service)
Cómputo (CPU/RAM)	$140 (Cluster/Nodos)	$55 (Plan B1/S1)
Base de Datos	$15 (Instancias compartidas)	$5 (Single DB)
Networking/Ingress	$30 (Load Balancer)	$0 (Incluido)
Observabilidad	$50 (Logs distribuidos)	$10 (Logs centralizados)
Total Mensual	**$235 USD**	**$70 USD**

Para una entidad pública municipal con presupuesto limitado, la diferencia es abismal. El ahorro del 70 % en infraestructura permite redirigir recursos hacia desarrollo evolutivo o capacitación de usuarios, aportando más valor real que una arquitectura "perfecta"pero insostenible.

La síntesis de los resultados obtenidos en la implementación de GESCOMPH plantea una reflexión crítica sobre la inercia tecnológica que a menudo empuja a las instituciones públicas hacia arquitecturas distribuidas prematuras. Si bien la literatura contemporánea, ejemplificada por trabajos como los de [14], ensalza las virtudes de la escalabilidad infinita de los microservicios, nuestra experiencia sugiere que para sistemas de gestión con una carga transaccional predecible y una complejidad de dominio moderada, el costo operativo de dicha distribución supera sus beneficios marginales. La adopción de una Arquitectura Limpia sobre un Monolito Modular no ha estado exenta de fricciones; la estricta inversión de dependencias introduce una verbosidad innegable en el código

—el llamado ”boilerplate”— y exige una disciplina cognitiva constante por parte del equipo de desarrollo para no sucumbir a la tentación de los atajos arquitectónicos. Sin embargo, este costo inicial se revela como una inversión en longevidad. Al contrastar nuestra estructura con la deuda técnica documentada por [18] en sistemas que priorizaron la velocidad de entrega sobre la integridad estructural, se hace evidente que la rigidez de nuestras fronteras físicas (`.csproj`) actúa como un seguro contra la entropía, permitiendo que el sistema mantenga su mantenibilidad a lo largo de los ciclos presupuestarios anuales típicos del sector público.

Más allá de la estabilidad actual, la discusión debe abordar la viabilidad evolutiva del sistema. Los críticos del enfoque monolítico podrían argumentar que GESCOMPH es vulnerable a cuellos de botella de escalabilidad en componentes específicos. Reconocemos esta limitación teórica; sin embargo, la estrategia de *Vertical Slicing* implementada neutraliza el riesgo de bloqueo arquitectónico. A diferencia de un monolito tradicional donde la lógica está entrelazada, nuestros módulos de ”Notificaciones.o .Auditoría”son candidatos triviales para una extracción futura hacia funciones *serverless* o contenedores independientes si la telemetría así lo indicara. Esta capacidad de diferir la decisión de distribución hasta el ”último momento responsable.es, en sí misma, una ventaja táctica que nos ha permitido centrar los recursos limitados en la corrección de la lógica de negocio y la seguridad, en lugar de en la orquestación de un enjambre de servicios. Mirando hacia el horizonte tecnológico, la evolución natural de GESCOMPH no apunta hacia una explosión de microservicios, sino hacia una integración más profunda de capacidades de inteligencia en los bordes del monolito; la incorporación de validaciones predictivas mediante IA o la inmutabilidad de registros mediante Blockchain, sugeridas por [5], son factibles precisamente porque el núcleo transaccional es sólido y coherente, no fragmentado. En última instancia, este estudio defiende que la verdadera modernización no reside en la adopción ciega de patrones de hiperescala, sino en la construcción de sistemas que sean lo suficientemente robustos para operar hoy y lo suficientemente modulares para cambiar mañana.

11.2. Lecciones Aprendidas y Errores Superados

El camino hacia esta arquitectura no fue lineal. Documentamos aquí los errores más significativos para beneficio de futuros implementadores:

- **La trampa de la abstracción prematura:** Inicialmente, intentamos desacoplar completamente la API de la lógica usando el patrón *Mediator* (librería MediatR). Si bien redujo el acoplamiento, hizo que la navegación por el código fuera una pesadilla (”Go to definition”llevaba al handler genérico, no a la implementación). **Solución:** Revertimos a inyección de dependencias directa para servicios de dominio, reservando MediatR solo para eventos de dominio asíncronos.
- **Infierno de Permisos en Docker:** El desarrollo en Windows con contenedores Linux presentó desafíos constantes de permisos de escritura en volúmenes montados (especialmente para la generación de reportes PDF y logs). **Solución:** Estandarizamos el uso de usuarios no-root dentro de los contenedores y configuramos explícitamente los GID/UID en el `docker-compose.yml`, una práctica que ahora es parte de nuestro estándar de seguridad.
- **Duplicidad de Validaciones:** Al principio, validábamos reglas de negocio (e.g., ”fecha fin >fecha inicio”) tanto en el Frontend (React) como en el Backend. Esto generaba inconsistencias cuando cambiaba una regla. **Solución:** Centralizamos las reglas en el Backend usando *FluentValidation* y expusimos un endpoint de ”Dry-Run”que el Frontend consulta para validar formularios complejos antes del envío final.

12. Conclusiones

Existe una inercia en la industria del software que empuja a los equipos a adoptar microservicios casi por defecto, equiparando distribución con modernidad. Nuestra experiencia construyendo GESCOMPH desafía esa noción. Al final del día, lo que hemos comprobado es que la complejidad de red es un precio demasiado alto para sistemas que, como este, requieren ante todo integridad transaccional y simplicidad operativa. No hemos construido un ”monolito.en el sentido peyorativo de código espagueti; hemos construido una unidad de despliegue única que, internamente, respeta fronteras tan estrictas como si fueran servicios físicos.

La lección crítica aquí no es sobre tecnología, sino sobre disciplina. Usar .NET 8 y Docker no garantiza nada si la arquitectura subyacente permite atajos. Lo que realmente ha marcado la diferencia en este proyecto ha sido la imposición de barreras físicas —los archivos `.csproj` separados— que obligaron al equipo a pensar en contratos e interfaces antes de escribir una sola línea de lógica. Esa fricción, que a veces resultaba molesta durante el desarrollo temprano, es la que ahora nos permite afirmar que el sistema es mantenible.

Mirando hacia adelante, GESCOMPH no necesita ser reescrito para escalar. Gracias al *Vertical Slicing*, ya tenemos las çosturas”por donde cortar el sistema si algún día el tráfico lo exige. Pero hasta que ese día llegue, nos quedamos con la eficiencia de la memoria compartida y la simplicidad de una sola base de datos. La verdadera innovación en el sector público no está en usar la arquitectura de Netflix, sino en entregar software que funcione, que sea seguro y que no requiera un ejército de ingenieros para mantenerlo operativo. Ese es el pragmatismo que defendemos, alineado con la visión de gobernanza inteligente de [20].

A. Esquema de Base de Datos Relacional

El modelo de datos de GESCOMPH se ha diseñado siguiendo la tercera forma normal (3NF) para garantizar la integridad referencial. A continuación se describen las entidades principales y sus relaciones:

Contract (Contrato): Entidad central del sistema.
- `Id (PK, int)`: Identificador único.
- `PersonId (FK, int)`: Referencia al arrendatario.
- `TotalBaseRentAgreed (decimal)`: Valor del canon pactado.
- `Active (bool)`: Estado del contrato.
- `StartDate / EndDate (datetime)`: Vigencia.

Establishment (Establecimiento): Unidad física arrendable.
- `Id (PK, int)`: Identificador único.

- `PlazaId (FK, int)`: Ubicación macro.
- `RentValueBase (decimal)`: Valor base del canon.
- `Active (bool)`: Disponibilidad.

PremisesLeased (Locales Arrendados): Tabla de relación N:M.

- `ContractId (FK, int)`
- `EstablishmentId (FK, int)`

User (Usuario): Autenticación y roles.

- `Id (PK, int)`
- `Email (varchar)`
- `PasswordHash (varchar)`
- `PersonId (FK, int, nullable)`

AuditLog (Auditoría): Registro inmutable.

- `Id (PK, long)`
- `TableName (varchar)`
- `Action (varchar)`
- `OldValues (jsonb)`
- `NewValues (jsonb)`
- `ChangedBy (varchar)`
- `ChangedAt (datetime)`

PaymentTransaction (Transacciones): Procesos financieros.

- `Id (PK, guid)`
- `ContractId (FK, int)`
- `Amount (decimal)`
- `Status (varchar)`
- `ProviderReference (varchar)`
- `PaymentDate (datetime)`

B. Especificación de API REST (Ejemplos)

B.1. Creación de Contrato (POST /api/contract)

Listing 12: Payload JSON para la creación de un contrato.

```
{
  "personId": 1054,
  "startDate": "2023-01-01T00:00:00Z",
  "endDate": "2023-12-31T23:59:59Z",
  "establishmentIds": [ 12, 14 ],
  "clauseIds": [ 1, 2, 5 ],
  "observations": "Contrato renovado con adición
      presupuestal."
}
```

B.2. Respuesta Exitosa (201 Created)

Listing 13: Respuesta JSON tras la creación exitosa.

```
{
  "id": 2045,
  "fullName": "Juan Perez",
  "totalBaseRentAgreed": 1500000.00,
  "active": true,
  "createdAt": "2023-10-27T10:30:00Z"
}
```

C. Estructura del Proyecto (.NET Solution)

```
GESCOMPH.sln
|-- GESCOMPH.Entity (Core)
|   |-- Domain
|   |   |-- Models
|   |   |   |-- Contract.cs
|   |   |   |-- Establishment.cs
|   |-- DTOs
|       |-- ContractCreateDto.cs
|
|-- GESCOMPH.Business (Logic)
|   |-- Interfaces
|   |   |-- IContractService.cs
|   |-- Services
|       |-- ContractService.cs
|       |-- TokenBusiness.cs
|
|-- GESCOMPH.Data (Infrastructure)
|   |-- Context
|   |   |-- ApplicationDbContext.cs
|   |-- Services
|       |-- ContractRepository.cs
|
|-- GESCOMPH.WebGESCOMPH (Presentation)
    |-- Controllers
    |   |-- ContractController.cs
    |-- Program.cs
    |-- appsettings.json
```

D. Configuración del Entorno

Tabla 5: Variables de Entorno Críticas

Variable	Descripción	Ejemplo (Oculto)
`ConnectionStrings__Default`	Cadena de conexión SQL Server	`Server=db;Database=ges...`
`JwtSettings__Key`	Clave secreta para tokens	`[REDACTED_256_BIT_KEY]`
`JwtSettings__Issuer`	Emisor del token	`gescomph-api`
`MercadoPago__AccessToken`	Token de pagos	`TEST-8493...`
`ASPNETCORE_ENVIRONMENT`	Entorno de ejecución	`Production`

Referencias

[1] Priyanka Billawa et al. «SoK: Security of Microservice Applications: A Practitioners' Perspective on Challenges and Best Practices». En: *arXiv preprint arXiv:2202.01612* (2022). URL: https://arxiv.org/pdf/2202.01612.

[2] Grzegorz J. Blinowski, Anna Ojdowska y Adam Przybyłek. «Monolithic vs. Microservice Architecture: A Performance and Scalability Evaluation». En: *IEEE Access* (2022). URL: https://ieeexplore.ieee.org/abstract/document/9717259/metrics.

[3] Maria Manuela Cruz-Cunha y Nuno Mateus-Coelho. «Security in Microservices Architectures». En: *Procedia Computer Science* (2020). URL: https://pdf.sciencedirectassets.com/280203/1-s2.0-S1877050921X0004X/1-s2.0-S1877050921003719/main.pdf.

[4] Aswinkumar Dhandapani. «Automation Testing in Microservices and Cloud-Native Applications: Strategies and Innovations». En: *Journal of Computer Science and Technology Studies (JCSTS)* (2025). URL: https://al-kindipublishers.org/index.php/jcsts/article/view/9682.

[5] Deepak Kaul. «Dynamic Adaptive API Security Framework Using AI-Powered Blockchain Consensus for Microservices». En: *International Journal of Scientific Research and Management (IJSRM)* (2020). URL: https://download.ssrn.com/2025/1/14/5096211.pdf.

[6] Sandeep Konakanchi. «Predictive Cyber-Resilience: AI-Powered Self-Defending Microservices for Zero-Downtime Security». En: *International Journal of Scientific Research in Computer Science, Engineering and Information Technology (IJSRCSEIT)* (2025). URL: http://prjcs.com/index.php/prjcs/article/view/67/68.

[7] Mounika Kothapalli. «Securing Microservices Architecture: Best Practices and Challenges». En: *Google Scholar* (2021). URL: https://www.researchgate.net/profile/Mounika-Kothapalli-2/publication/384076350_Securing_Microservices_Architecture_Best_Practices_and_Challenges/links/66fac862553d2145f9e4501c/Securing-Microservices-Architecture-Best-Practices-and-Challenges.pdf.

[8] Aditya Manikyala. «Integrating Cybersecurity Best Practices in DevOps Pipelines for Securing Distributed Systems». En: *Google Scholar* (2023). URL: https://www.researchgate.net/profile/Aditya-Manikyala/publication/385893283_Integrating_Cybersecurity_Best_Practices_in_DevOps_Pipelines_for_Securing_Distributed_Systems/links/673a195537496239b2c8f85d/Integrating-Cybersecurity-Best-Practices-in-DevOps-Pipelines-for-Securing-Distributed-Systems.pdf.

[9] Manuel Mazzara y A. B. M. Moniruzzaman. «Benchmarks and performance metrics for assessing the migration to microservice-based architectures». En: *ResearchGate* (2024). URL: https://www.researchgate.net/profile/Manuel-Mazzara/publication/353324227_Migration_from_Monolith_to_Microservices_Benchmarking_a_Case_Study/links/60ff050b1e95fe241a8e66eb/Migration-from-Monolith-to-Microservices-Benchmarking-a-Case-Study.pdf.

[10] Manuel Mazzara, A. B. M. Moniruzzaman y Dilshod Kuryazov. «Migration from Monolith to Microservices: Benchmarking a Case Study». En: *ResearchGate* (2020). URL: https://www.researchgate.net/profile/Manuel-Mazzara/publication/339749917_Migration_from_Monolith_to_Microservices_Benchmarking_a_Case_Study/links/5e6359034585153fb3c8515f/Migration-from-Monolith-to-Microservices-Benchmarking-a-Case-Study.pdf.

[11] Teemu Myllynen et al. «Developing a Conceptual Model for Cross-Domain Microservices Using Event-Driven and Domain-Driven Design». En: *International Journal of Multidisciplinary Research and Growth Evaluation* (2023). URL: https://www.researchgate.net/publication/388554873_Developing_a_Conceptual_Model_for_Cross-Domain_Microservices_Using_Event-Driven_and_Domain-Driven_Design.

[12] Nivedhaa N. «SOFTWARE ARCHITECTURE EVOLUTION: PATTERNS, TRENDS, AND BEST PRACTICES». En: *ResearchGate* (2024). URL: https://www.researchgate.net/profile/Nivedhaa-N/publication/384019495_SOFTWARE_ARCHITECTURE_EVOLUTION_PATTERNS_TRENDS_AND_BEST_PRACTICES/links/66e52757fa5e11512cb89d26/SOFTWARE-ARCHITECTURE-EVOLUTION-PATTERNS-TRENDS-AND-BEST-PRACTICES.pdf.

[13] Venkata Durga Ganesh Nandigam. «CHAOS ENGINEERING: STRESS-TESTING MICROSERVICES FOR RESILIENCE». En: *Google Scholar* (2024). URL: https://www.researchgate.net/profile/Venkata-Durga-Ganesh-Nandigam-2/publication/389674431_Chaos_engineering_Stress-testing_microservices_for_resilience/links/6840500bc33afe388aca1535/Chaos-engineering-Stress-testing-microservices-for-resilience.pdf.

[14] Oyekunle Claudius Oyeniran et al. «Microservices Architecture in Cloud-Native Applications: Design Patterns and Scalability». En: *International Journal of Applied Research in Science and Engineering (IJARISE)* (2024). URL: https://ijarise.org/index.php/ijarise/article/view/66.

[15] Sridhar Patlolla. «Architectural Evolution from Monolithic to Microservices in Scalable Systems: A Case Study of Netflix». En: *Dialnet* (2025). URL: https://dialnet.unirioja.es/servlet/articulo?codigo=10164353.

[16] Sridhar Patlolla. «Enhancing Security: Vulnerability Detection and Monitoring in Microservices within Multi-Cloud Environments Utilizing Sysdig». En: *Primera Scientific Engineering (PSEN)* (2022). URL: https://primerascientific.com/publicaciones/en/PSEN-06-178.pdf.

[17] Basavegowda Ramu. «Performance Impact of Microservices Architecture». En: *The Review of Contemporary Scientific and Academic Studies (RCSAS)* (2023). URL: https://thercsas.com/wp-content/uploads/2023/06/rcsas3062023010.pdf.

[18] Saulo Soares de Toledo et al. «Architectural Technical Debt in Microservices: A Case Study in a Large Company». En: *IEEE/ACM International Conference on Technical Debt (TechDebt)* (2019). URL: https://d1wqtxts1xzle7.cloudfront.net/112276296/TechDebt_2019_paper___ATD_in_MS___PREPRINT-libre.pdf.

[19] Victor Velepucha y Pamela Flores. «A Survey on Microservices Architecture: Principles, Patterns and Migration Challenges». En: *IEEE Access* (2023). URL: https://ieeexplore.ieee.org/abstract/document/10220070.

[20] Indra Yustiana, Hermanto Hermanto y Fitrina Lestari. «Microservices in strategic public service delivery: a systematic review and future research agenda for smart governance». En: *Digital Business: Tren Bisnis Masa Depan* (2025). URL: https://ejournal.cria.or.id/index.php/db/article/view/301/235.