Documentación Técnica – Provider Optimizer

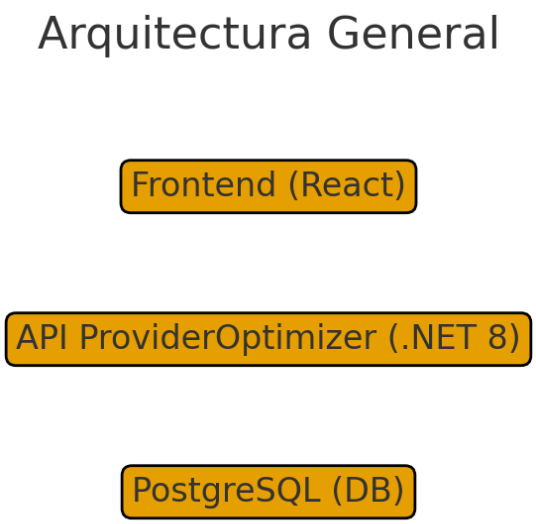
SOLUCIÓN A PRUEBA TÉCNICA — LÍDER TÉCNICO DE DESARROLLO (ASISYA)

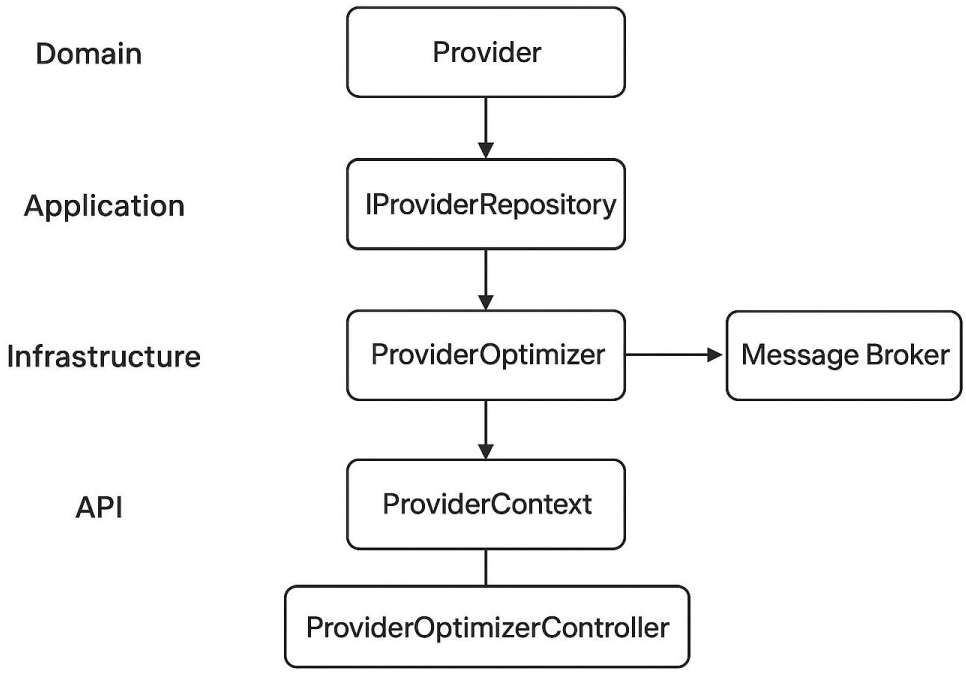
Versión con arquitectura, diagramas, estándares técnicos y mitigación de Riesgos que se tomó en cuenta

1. Objetivo del Proyecto

Provider Optimizer es un microservicio diseñado para seleccionar el proveedor óptimo de asistencia según ubicación, tipo de servicio y calificación en la que también muestra el seguimiento en tiempo real del estado de la solicitud. Incluye un frontend en React y una API en .NET 8 conexión a PostgreSQL.

2. Arquitectura de la Solución





La arquitectura se basa en una solución moderna con separación clara por capas.

- Backend: Microservicio .NET 8 ProviderOptimizer.API
- Frontend: React + Axios
- Base de datos: PostgreSQL
- Orquestación: Docker & Docker Compose
- Patrón: Clean Architecture y Minimal API.

Para Ver propuesta inicial revisar el anexo architecture.md

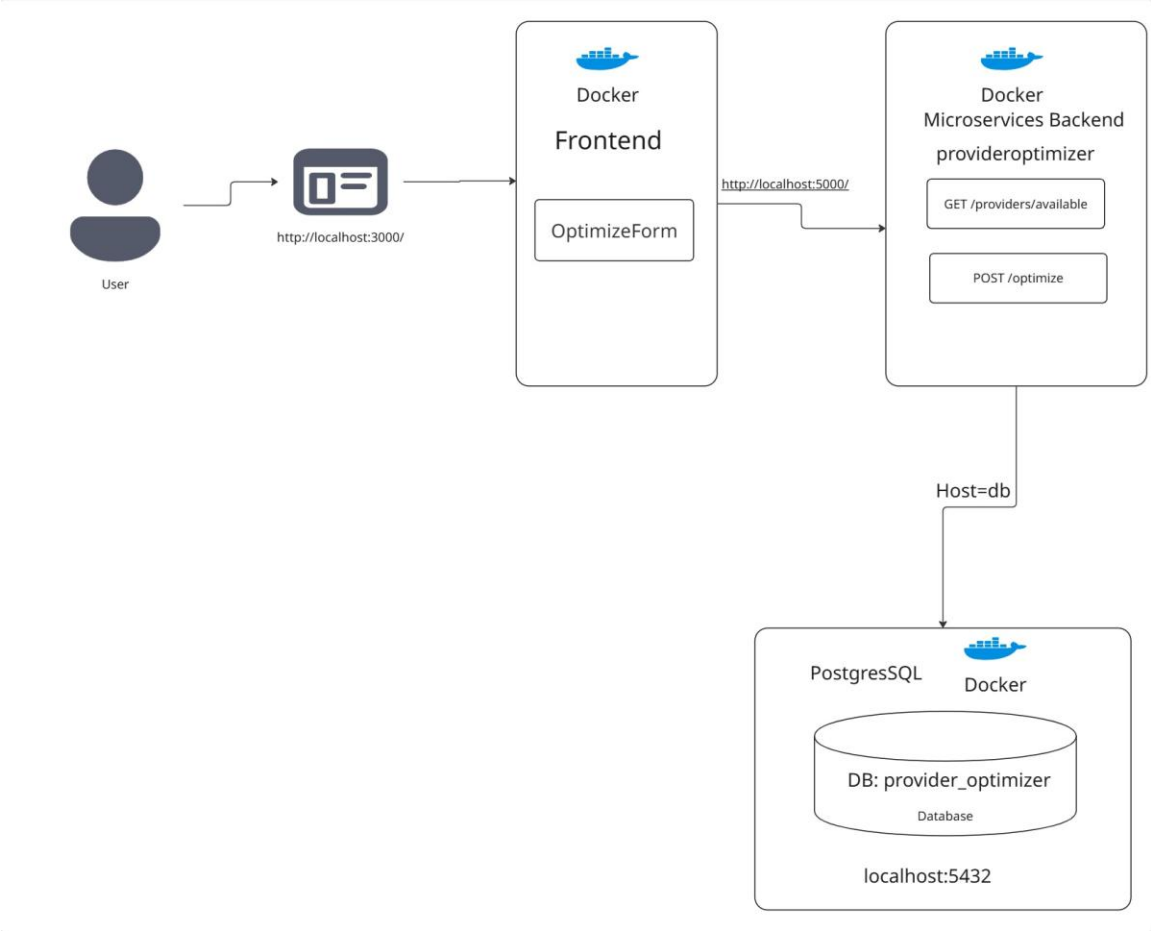
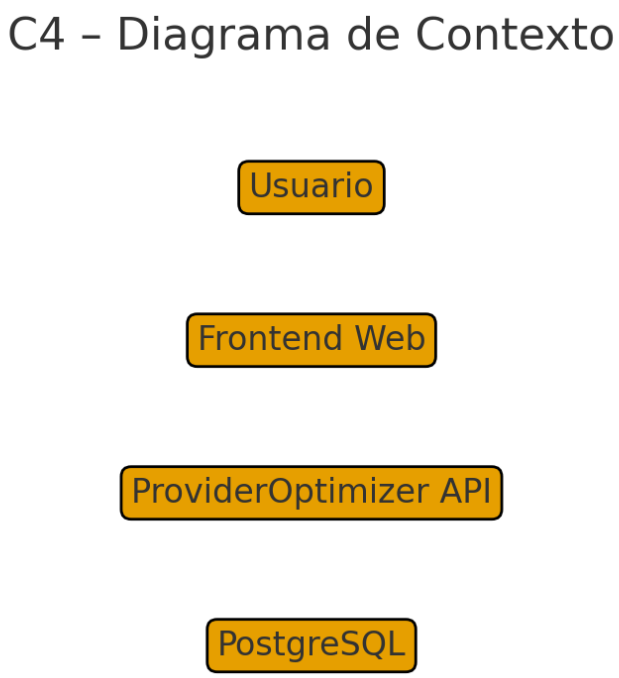
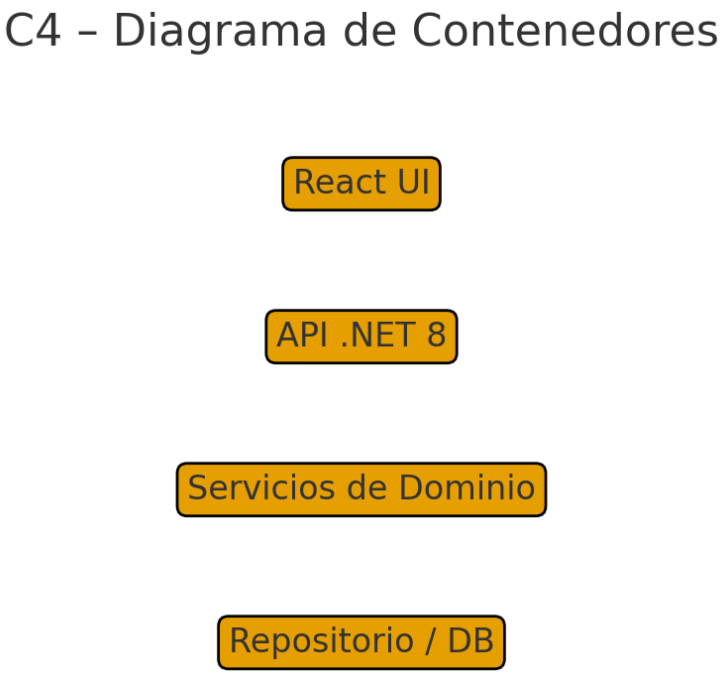


Diagrama Arquitectura

3. Diagrama C4 – Contexto



4. Diagrama C4 – Contenedores



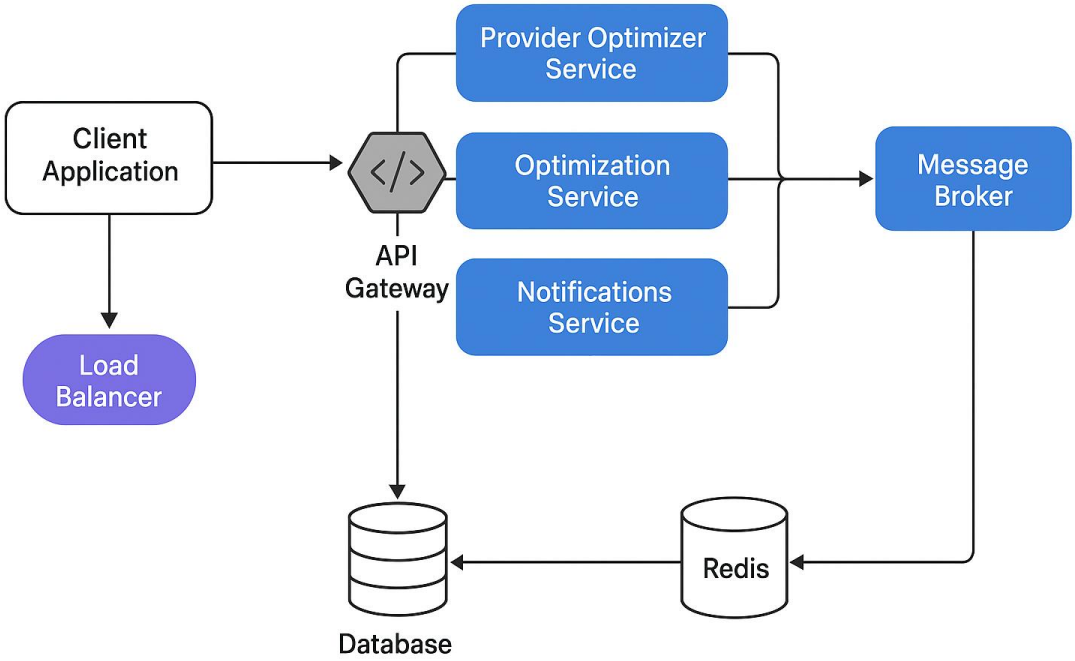


Diagrama Secuencia

5. Stack Tecnológico

Tecnología	Uso
React + Vite	Frontend profesional UI/UX
.NET 8 Minimal API	Microservicio principal
PostgreSQL	Base de datos relacional
Docker / Docker Compose	Orquestación de entornos
Axios	Consumo REST desde frontend
EF Core	ORM para acceso a datos

6. Endpoints del Microservicio

GET http://localhost:5000/providers/available

POST http://localhost:5000/optimize

GET http://localhost:5000/tracking/{requestId}

Base URL: http://localhost:5000

7. Instalación y Ejecución

- 1. Clonar repositorio: https://github.com/angelicadr/provider-optimizer.git
  2. Ejecutar: docker compose up --build
  3. Frontend disponible en: http://localhost:3000
  4. API disponible en: http://localhost:5000/swagger
  5. dotnet test → ejecuta todas las pruebas.

8. Estándares Técnicos Implementados

- Clean Architecture (capas: Dominio, Aplicación, Infraestructura, API)
- Buenas prácticas REST
- CORS configurado
- Validación y DTOs
- Logs estructurados

Ver más sobre el estándar en standards.md

9. Code Review – Mejoras aplicadas

- Corrección de CORS completo
- Manejo adecuado de errores HTTP
- Refactorización de servicios y repositorios
- Limpieza de Program.cs usando Minimal API

ver más sobre code Review en code_review.md y en code_review_response.md

10. Modelo de datos

Modelo entidad relación de la base de datos provider_optimizer

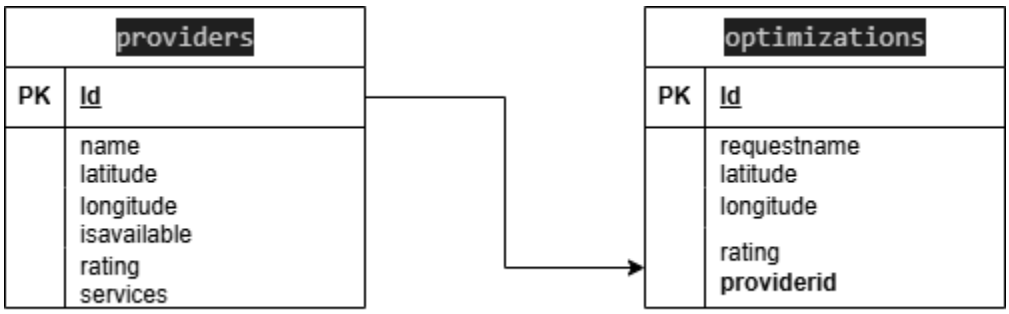


Diagrama Base de Datos

Riesgos mitigados en la solución implementada

La arquitectura y el desarrollo del microservicio **Provider Optimizer** contemplan varios riesgos comunes en soluciones empresariales. Estos son los principales **riesgos mitigados**:

1. Riesgo de indisponibilidad de la base de datos

Mitigación:

- Uso de docker-compose con healthchecks.
- El microservicio espera a que PostgreSQL esté "healthy" antes de iniciar.
- Conexiones parametrizadas vía ConnectionStrings__DefaultConnection.

✓ Evita fallas en tiempo de inicio y errores por conexión rechazada.

2. Riesgo de bloqueo de solicitudes por CORS

Mitigación:

- Implementación explícita de política CORS con origen permitido (http://localhost:3000).
- Métodos, headers y credenciales habilitadas adecuadamente.

✓ Garantiza comunicación segura entre frontend y backend.

3. Riesgo de errores por incompatibilidad entre ambientes

Mitigación:

- Uso de Docker multi-stage para compilación y ejecución.
- Configuración separada por environment (Development, Production).
- Variables de ambiente controladas.

✓ Minimiza diferencias entre máquinas locales y servidores.

4. Riesgo de degradación por lógica de selección de proveedores

Mitigación:

- Servicios desacoplados (Domain → Application → Infrastructure).
- Repositorio con consultas eficientes.
- Filtros por disponibilidad, tipo de servicio y rating.

✓ Algoritmos resistentes a datos inconsistentes o incompletos.

5. Riesgo de errores en la validación de entrada

Mitigación:

- Tipos estrictos (double, int, enums).
- DTO validado antes de procesar.
- Manejo seguro de errores en el frontend.

✓ Evita inyecciones, valores inválidos y solicitudes corruptas.

6. Riesgo de acoplamiento entre módulos

Mitigación:

- Arquitectura por capas.
- Dependency Injection para repositorios y servicios.
- Interfaces (IProviderRepository, IProviderOptimizerService).

✓ Facilita mantenimiento, pruebas unitarias y escalabilidad

7. Riesgo de código defectuoso

Mitigación:

- Se incluye *Code Review* con hallazgos y correcciones.
- Refactor del snippet defectuoso.
- Lineamientos de estándares técnicos del squad.

✓ Fomenta calidad, detecta errores lógicos y malas prácticas.

8. Riesgo de documentación insuficiente

Mitigación:

- Swagger generado.
- Documentación técnica entregada.
- Diagramas (ERD, arquitectura, C4).

✓ Reduce curva de aprendizaje y facilita soporte.