{ }

MIGRACIÓN DE APLICACIONES CON IA

Serie IA Práctica · Leandro Sepúlveda

Cómo Migrar Aplicaciones Legacy con Agentes de IA

Una guía general, sin atajos, para migrar cualquier stack legacy usando agentes especializados

Leandro Juan Sepúlveda Cuevas
Software Engineer · leandrosepulveda.com

2026

Cómo Migrar Aplicaciones Legacy con Agentes de IA

Guía general, sin atajos, para migrar cualquier stack legacy usando agentes especializados

Autor: Leandro Juan Sepúlveda Cuevas
Software Engineer — leandrosepulveda.com

Copyright © 2026 Leandro Juan Sepúlveda Cuevas. Todos los derechos reservados.
Queda prohibida la reproducción total o parcial de esta obra sin la autorización expresa del autor.
Primera edición: 2026. Segunda edición revisada y ampliada: 2026.

Este libro enseña un **método general** de migración de aplicaciones legacy con agentes de IA, aplicable a cualquier combinación de lenguaje y framework de origen y destino. Usamos PHP/Laravel a .NET como caso guía porque es una migración muy común en 2026, pero el método —auditar, mapear equivalencias, definir agentes y skills, migrar módulo por módulo, testear y desplegar gradualmente— aplica igual si tu caso es Java a Kotlin, Python 2 a Python 3, o cualquier otro par de tecnologías. Los nombres de herramientas mencionados (PHP, Laravel, .NET, OpenCode, Ollama, Groq, OpenRouter, etc.) pertenecen a sus respectivos dueños, y los comandos y modelos disponibles cambian seguido: confirma los detalles puntuales en la documentación oficial de cada herramienta antes de correrlos en un proyecto real.

Contenido

01 Por que migrar y por que ahora

Cuando migrar deja de ser un lujo y pasa a ser una necesidad

02 El método general de migración asistida

Los cinco pasos que aplican a cualquier stack

03 Entendiendo el terreno

Como analizar tu código legacy antes de tocar nada

04 La arquitectura objetivo

Clean Architecture, equivalencias y el archivo de agentes

05 El Skill de migración

Enseñándole al agente a convertir un lenguaje en otro

06 Migrando módulo por módulo

El orden recomendado y los casos difíciles

07 Autenticación y sesiones

El módulo que más dolores de cabeza da en cualquier migración

08 Archivos, storage e integraciones de terceros

Lo que vive fuera de la base de datos

09 Testing de la migración

Por que los tests importan más acá que en cualquier otro proyecto

10 CI/CD para el proyecto nuevo

Automatizar build, test y despliegue desde el primer módulo

11 Puesta en producción

Estrategia Strangler Fig y plan de rollback

12 Glosario y recursos

Los términos que vas a escuchar todo el tiempo en este proceso

Por que migrar y por que ahora

CAPÍTULO 01

Si estas leyendo esto es porque tienes una aplicación legacy que ya pesa. Quizás arranco como un proyecto chico y ahora tiene 50 mil líneas, 30 tablas en la base de datos, y nadie se anima a tocarla porque funciona pero no se sabe bien como ni por que.

Migrar una aplicación de un stack a otro no siempre es un lujo —a veces es una necesidad real. Tal vez necesitas mejor performance, escalar horizontalmente, integrarte mejor con otro ecosistema, o simplemente no consigues desarrolladores que quieran hacerse cargo de mantener una tecnología vieja.

La novedad de los últimos años es que ya no tienes que hacer la migración a mano, archivo por archivo, durante seis meses. Ahora tienes agentes de IA especializados que entienden tanto el lenguaje de origen como el de destino, que pueden leer tu código legacy y generar el equivalente en el stack nuevo, siempre con revisión humana en el medio.

Este libro te lleva paso a paso por el método completo de migrar una aplicación legacy usando agentes y skills especializados. El caso guía que usamos de principio a fin es una migración de Laravel (PHP) a .NET 8 con OpenCode, aprovechando que hoy puedes usar modelos gratuitos o muy baratos como Mistral, DeepSeek o Qwen para gran parte del trabajo —pero cada capítulo explica el principio general antes del ejemplo puntual, para que puedas aplicar exactamente el mismo método a tu propio par de tecnologías.

> **UNA ACLARACIÓN HONESTA**
>
> Ningún agente de IA te va a dar una migración perfecta con solo apretar un botón. Lo que si te va a dar es una aceleración enorme del trabajo mecánico y repetitivo, para que tú dediques tu tiempo a revisar, corregir y validar —que es donde realmente se necesita criterio humano.

El método general de migración asistida

CAPÍTULO 02

Antes de entrar en el caso concreto de PHP a .NET, conviene dejar clarísimo el método en abstracto, porque es lo que te vas a llevar de este libro sin importar tu stack real.

1. **Auditar antes de tocar nada:** ningún agente debería empezar a escribir código nuevo sin primero entender la estructura completa del sistema origen —módulos, dependencias, patrones, calidad del código (capítulo 3).
2. **Definir la arquitectura objetivo y el mapa de equivalencias:** antes de migrar la primera línea, decide cómo se va a organizar el código nuevo y arma una tabla de "esto en el origen equivale a esto en el destino" (capítulo 4).
3. **Enseñarle el patrón al agente con un Skill:** un documento estructurado que describe exactamente como convertir cada construcción típica del lenguaje origen a su equivalente en el destino (capítulo 5).
4. **Migrar de a módulos, en el orden correcto:** nunca migres todo de una vez. Empieza por los módulos más independientes y sigue una secuencia que respete las dependencias reales (capítulos 6, 7 y 8).
5. **Testear y desplegar gradualmente:** cada módulo migrado se prueba a fondo antes de exponerlo a trafico real, y el sistema viejo sigue funcionando en paralelo hasta que la migración completa esta probada (capítulos 9, 10 y 11).

> **POR QUE ESTE ORDEN IMPORTA**
>
> Este método es idéntico si estas migrando Java a Kotlin, un monolito Rails a microservicios, o Python 2 a Python 3. Lo único que cambia entre un caso y otro es el contenido del Skill del paso 3 y la tabla de equivalencias del paso 2 —el resto del proceso es el mismo esqueleto.

Entendiendo el terreno

CAPÍTULO 03

Antes de migrar, tienes que entender que tienes

El error más grande que puedes cometer es agarrar el proyecto legacy y pedirle a una IA que lo pase al stack nuevo sin darle más contexto. Primero necesitas entender que módulos tiene, como se relacionan entre si, que patrones usa, y cual es el estado real del código —no el que crees que tiene, el que tiene de verdad.

Instalando OpenCode con modelos gratuitos

No necesitas pagar por APIs caras para arrancar. Puedes usar modelos abiertos gratis o casi gratis:

```
# Opción 1: Ollama (modelos locales, 100% gratis, corren en tu maquina)
winget install ollama # o el instalador correspondiente a tu sistema
ollama pull mistral
ollama pull deepseek-coder
opencode provider add ollama --base-url http://localhost:11434
opencode provider set-default ollama/mistral

# Opción 2: OpenRouter (tier gratuito limitado, buena variedad de modelos)
opencode provider add openrouter --model deepseek/deepseek-chat-v3

# Opción 3: Groq (muy rápido, tier gratuito con limite de peticiones por minuto)
opencode provider add groq --model llama3-70b-8192
```

Creando el Skill de análisis

Vamos a crear el skill que le enseña al agente como analizar un proyecto legacy de punta a punta —el mismo esqueleto sirve sin importar el lenguaje de origen, solo cambia el detalle de que archivos buscar:

```
mkdir -p .opencode/skills/legacy-analysis
cat > .opencode/skills/legacy-analysis/SKILL.md << 'EOF'
---
name: legacy-analysis
description: Analiza proyectos legacy para preparar la migracion
---
# Legacy Analysis Skill

Pasos:
1. Descubrir la estructura del proyecto
   - Leer el archivo de dependencias (composer.json, pom.xml, requirements.txt, etc.)
   - Listar directorios principales
   - Identificar archivos clave

2. Identificar módulos
   - Controllers/handlers, Models/entidades, Rutas, Migraciones de base de datos, Tests

3. Por cada módulo:
   - Número de archivos y líneas
   - Dependencias con otros módulos
   - Patrones usados
   - Calidad general del código
   - Prioridad de migración sugerida

4. Generar un reporte JSON con:
   - Módulos encontrados y su prioridad
   - Dependencias entre módulos
   - Recomendaciones y advertencias
EOF
```

Ejecutando la auditoría

```
opencode run --agent auditor --skill legacy-analysis \
    --input ./mi-app-legacy --output ./reports/analisis

cat ./reports/analisis/analisis-completo.json | python -m json.tool
```

El resultado te muestra algo parecido a esto (el ejemplo usa una app Laravel, pero el formato del reporte es igual para cualquier stack de origen):

```
{
  "framework": "Laravel 10",
  "modulos": [
    {"nombre": "Auth", "prioridad": "alta", "archivos": 8},
    {"nombre": "Usuarios", "prioridad": "alta", "dependencias": ["Auth"]},
    {"nombre": "Productos", "prioridad": "media", "complejidad": 7}
  ]
}
```

> **TIP**
>
> Guarda este reporte de análisis en el repositorio (no lo descartes). Vas a volver a el varias veces durante la migración, y sirve como documentación del estado original del sistema para quien venga después tuyo.

La arquitectura objetivo

CAPÍTULO 04

Clean Architecture, sin importar el destino

Clean Architecture es simplemente una forma de organizar el código para que sea mantenible en el tiempo, y aplica sin importar el lenguaje destino. La estructura típica se ve así:

```
mi-app-nueva/
├── MiApp.sln
├── src/
│   ├── Domain/         # Entidades y reglas de negocio
│   ├── Application/    # Casos de uso (Commands/Queries)
│   ├── Infrastructure/ # Base de datos, APIs externas
│   └── Web/            # Controladores, endpoints
└── tests/
    └── MiApp.Tests/
```

Mapeo de equivalencias: la guía central de tu migración

Esta tabla es el corazón de todo el proceso. Cada vez que tengas dudas sobre cómo convertir algo, vuelve acá. El ejemplo usa PHP/Laravel a .NET, pero la lógica —armar tu propia tabla de equivalencias antes de migrar— es lo que realmente importa copiar a tu propio caso:

PHP / Laravel	C# / .NET
Eloquent Model	Entity + IEntityTypeConfiguration
Controller	Controller o Minimal API Endpoint
FormRequest + validate()	FluentValidation Validator
Blade Template	Razor View o Blazor Component
Service Provider	IServiceCollection Extensión
Facade	Inyección de Dependencias
Eloquent::find()	_db.Set<T>().FindAsync()
where('x', $val)	.Where(e => e.X == val)
Model::create($data)	_db.Set<T>().AddAsync(entity)
Event + Listener	MediatR INotification + Handler
PHPUnit Test	xUnit Test + Moq + FluentAssertions

.env	appsettings.json + User Secrets
Migrations (php artisan)	dotnet ef migrations add

Creando el archivo de definición de agentes

El archivo de agentes (en OpenCode, `AGENTS.md`) define como los distintos agentes trabajan dentro del proyecto, que rol tiene cada uno y con que reglas —este patrón aplica a cualquier par de tecnologías, solo cambian los modelos y las instrucciones puntuales:

```
# AGENTS.md
agents:
  auditor:
    role: "Analista de codigo legacy"
    skill: legacy-analysis
    model: deepseek/deepseek-chat-v3
  migrator:
    role: "Migrador de Codigo"
    skill: dotnet-migration
    model: openrouter/deepseek/deepseek-chat-v3
    instructions:
      - "Convertir archivos del stack origen al stack destino"
      - "Seguir Clean Architecture"
      - "Correr el build del proyecto destino y corregir errores antes de terminar"
  tester:
    role: "Generador de Tests"
    skill: dotnet-testing
    model: ollama/mistral

rules:
  - Los agentes se pasan contexto entre si mediante archivos JSON
  - NUNCA modificar los archivos del proyecto origen
  - SIEMPRE ejecutar el build del proyecto destino antes de dar una tarea por terminada
```

> **NOTA**
>
> La regla de "nunca modificar los archivos del proyecto origen" no es un capricho: mientras dure la migración, la app original sigue siendo tu sistema en producción. Tocarla por accidente puede romper algo que hoy funciona perfecto.

El Skill de migración

CAPÍTULO 05

Enseñándole al agente a convertir un lenguaje en otro

Este es el skill más importante de todo el libro. Le enseña al agente exactamente como convertir cada construcción típica del lenguaje origen a su equivalente en el destino, con ejemplos concretos:

```
# skills/dotnet-migration/SKILL.md
---
name: dotnet-migration
description: Convierte modulos del stack origen a C# con Clean Architecture
---
## Reglas generales
1. Cada modulo se convierte en 4 capas:
   - Domain: entidades + value objects
   - Application: commands + handlers + validators
   - Infrastructure: DbContext + repositories
   - Web: endpoints Minimal API
2. Convenciones de codigo:
   - File-scoped namespaces (namespace X.Y;)
   - Primary constructors donde tenga sentido
   - record para DTOs y commands
   - Async/await para toda operacion de I/O
3. Validacion: usar FluentValidation siempre, nunca validacion manual dispersa

## Equivalencia de ejemplo: Eloquent Model -> Entity C#
class User extends Authenticatable {
    use HasFactory, Notifiable;
    protected $fillable = ['name', 'email'];
}

se convierte en:

public sealed class User : IdentityUser<Guid> {
    private User() {}
    public string Name { get; private set; }
    public static User Create(string name, Email email) {
        return new User { Id = Guid.NewGuid(), Name = name, Email = email.Value };
    }
}
```

Ejecutando la migración del primer módulo

```
# Migrar capa Domain
opencode run --agent migrator --skill dotnet-migration \
    --prompt "Migra el modulo Auth a Domain. Crea la entidad User."

# Migrar Application
opencode run --agent migrator \
    --prompt "Migra AuthController a Commands/Queries con MediatR"

# Migrar Infrastructure y Web en paralelo
opencode run --agent migrator --id infra-auth \
    --prompt "Crea Infrastructure/Auth: DbContext, UserConfiguration" &
opencode run --agent migrator --id web-auth \
    --prompt "Crea Web/Auth: endpoints Minimal API con JWT" &
wait

# Verificar que compile y pase los tests
dotnet build
dotnet test
```

Modelos gratuitos que funcionan bien para esto

- **DeepSeek V3 (via OpenRouter):** muy buen resultado en tareas de código, con tier gratuito generoso.
- **Llama 3.x 70B (via Groq):** rápido de respuesta, con un límite de peticiones por minuto en el plan gratis.
- **Mistral 7B (via Ollama):** corre 100% local, sin costo de API, ideal si tus datos no pueden salir de tu red.
- **Qwen 2.5/3 Coder (via Ollama):** especializado en código, también corre local.

> **ATENCIÓN**
>
> Los modelos locales chicos (7B) son más lentos y menos precisos que un modelo de frontera pago. Para migraciones moderadamente complejas funcionan bien igual, pero para los módulos más críticos de tu sistema conviene invertir en un modelo más grande y revisar el resultado con más cuidado todavía.

Migrando módulo por módulo

CAPÍTULO 06

El orden de migración recomendado

1. **Auth:** el módulo más independiente, establece la base de Identity para todo lo demás (ver capítulo 7, dedicado entero a este módulo por lo delicado que es).
2. **Usuarios:** depende de Auth, lógica de CRUD relativamente simple.
3. **Productos:** lógica de negocio ya más compleja.
4. **Ordenes:** normalmente el módulo más complejo, depende de Usuarios y Productos a la vez.
5. **Reportes:** mayormente lógica de consultas, buen candidato para el final del proceso.

Script de migración completa

```
#!/bin/bash
MODULOS=("Auth" "Usuarios" "Productos" "Ordenes" "Reportes")

for modulo in "${MODULOS[@]}"; do
    echo "Migrando: $modulo"
    opencode run --agent auditor --skill legacy-analysis \
        --prompt "Analiza el modulo $modulo"
    opencode run --agent migrator --skill dotnet-migration \
        --prompt "Migra el modulo $modulo completo"
    opencode run --agent tester --skill dotnet-testing \
        --prompt "Genera tests para el modulo $modulo"
    echo "$modulo completado"
done

dotnet build && dotnet test
```

Lidiando con los casos difíciles

SQL directo sin ORM: el agente necesita migrar consultas SQL crudas a EF Core o a Dapper. Se lo más específico posible en el prompt: "Usa Dapper para queries complejas, EF Core para CRUD estándar".

Templates complejos: primero migra la lógica a un handler o service en el lenguaje destino, y después crea una API que el frontend consuma —no intentes convertir una vista compleja de una sola pasada.

APIs externas sin documentación: ejecuta el código original con logging activado para capturar las llamadas reales que hace. Con esos logs, el agente puede generar clientes HTTP en el lenguaje destino basados en el trafico real observado, en vez de adivinar.

Autenticación y sesiones

CAPÍTULO 07

Le dedicamos un capítulo entero a este módulo porque, en cualquier migración, es el que más dolores de cabeza da. Un error acá no es solo un bug: puede significar usuarios que quedan afuera del sistema, sesiones que se pisan entre sí, o una brecha de seguridad.

El problema central: dos formas distintas de "recordar" a un usuario

Los frameworks legacy basados en PHP suelen manejar sesiones del lado del servidor: el usuario tiene una cookie con un identificador, y el servidor guarda el estado de esa sesión en archivos o en una base de datos. Los stacks modernos tienden a preferir tokens sin estado (JWT), donde toda la información necesaria viaja firmada dentro del propio token, sin que el servidor tenga que guardar nada. Migrar de un modelo al otro no es solo "cambiar la librería" —cambia la arquitectura misma de cómo se autentica cada request.

Estrategia de migración recomendada

1. **Fase de convivencia:** el sistema nuevo acepta tanto las cookies de sesión viejas como los JWT nuevos, validando contra el mismo origen de verdad (la base de datos de usuarios) durante toda la transición.
2. **Migración de contraseñas sin exponer nada:** nunca migres contraseñas en texto plano. Si el hash del sistema origen es compatible (por ejemplo bcrypt, usado tanto en Laravel como en .NET con las librerías correctas), puedes validar contra el hash existente sin pedirle al usuario que cambie su contraseña.
3. **Emisión de tokens nuevos en el login:** cuando un usuario inicia sesión exitosamente en el sistema nuevo, ahí sí se le emite un JWT, y de a poco los clientes (web, móvil) van migrando a usar ese token.
4. **Apagado del sistema de sesiones viejo:** solo cuando el 100% del tráfico usa el token nuevo, se puede dar de baja el mecanismo de sesión original.

Verificando compatibilidad de hashes de contraseña

```
// C# - Verificar un hash bcrypt generado originalmente por Laravel
using BCrypt.Net;

public bool ValidarPasswordLegacy(string passwordIngresado, string hashGuardado)
{
    // Laravel usa bcrypt por defecto (password_hash con PASSWORD_BCRYPT)
    // BCrypt.Net-Next es compatible con el mismo formato de hash
    return BCrypt.Net.BCrypt.Verify(passwordIngresado, hashGuardado);
}
```

> **ATENCIÓN**
>
> Si el sistema origen usa un algoritmo de hash débil o propio (MD5, SHA1 sin salt), no migres ese hash tal cual: es una oportunidad para forzar un cambio de contraseña seguro en el primer login post-migración, en vez de arrastrar una debilidad de seguridad al sistema nuevo.

Emitiendo JWT en el sistema nuevo

```
// C# - Generar un JWT al validar login
using System.IdentityModel.Tokens.Jwt;
using Microsoft.IdentityModel.Tokens;
using System.Security.Claims;
using System.Text;

public string GenerarToken(Usuario usuario, string claveSecreta)
{
    var claims = new[]
    {
        new Claim(JwtRegisteredClaimNames.Sub, usuario.Id.ToString()),
        new Claim(JwtRegisteredClaimNames.Email, usuario.Email),
    };

    var key = new SymmetricSecurityKey(Encoding.UTF8.GetBytes(claveSecreta));
    var credenciales = new SigningCredentials(key, SecurityAlgorithms.HmacSha256);

    var token = new JwtSecurityToken(
        issuer: "mi-app",
        audience: "mi-app-clientes",
        claims: claims,
        expires: DateTime.UtcNow.AddHours(2),
        signingCredentials: credenciales
    );

    return new JwtSecurityTokenHandler().WriteToken(token);
}
```

Migrando roles y permisos

No olvides el sistema de autorización (qué puede hacer cada usuario), que suele vivir pegado a la autenticación pero es un concepto distinto. Arma una tabla de equivalencia de roles y permisos igual que hiciste con el código (capítulo 4), y verifica con tests específicos que un usuario con rol "editor" en el sistema viejo tenga exactamente los mismos permisos en el sistema nuevo — un error de permisos de más es tan grave como uno de menos.

Archivos, storage e integraciones de terceros

CAPÍTULO 08

Lo que vive fuera de la base de datos

Una migración no termina en el código y la base de datos relacional. Casi toda aplicación real tiene archivos subidos por usuarios, integraciones con servicios de pago, envío de emails, y otros servicios externos que también necesitan un plan de migración propio.

Archivos y storage

Si tu aplicación legacy guarda archivos en el disco del servidor, esa es una de las primeras cosas que conviene desacoplar, migrando a un almacenamiento de objetos (compatible con S3, por ejemplo) antes incluso de tocar el resto del código. Esto te permite que ambos sistemas —el viejo y el nuevo, corriendo en paralelo durante la transición— lean y escriban archivos del mismo lugar, sin duplicar ni perder nada.

```
// C# - Cliente de storage compatible con S3
using Amazon.S3;
using Amazon.S3.Model;

public class ArchivoStorageService
{
    private readonly IAmazonS3 _s3Client;
    private readonly string _bucket;

    public async Task<string> SubirArchivoAsync(Stream contenido, string nombreArchivo)
    {
        var request = new PutObjectRequest
        {
            BucketName = _bucket,
            Key = $"uploads/{nombreArchivo}",
            InputStream = contenido,
        };
        await _s3Client.PutObjectAsync(request);
        return $"https://{_bucket}.s3.amazonaws.com/uploads/{nombreArchivo}";
    }
}
```

Integraciones de terceros: pagos, email y más

Cada integración externa (pasarela de pagos, proveedor de email transaccional, servicio de mensajería) necesita su propio mini-plan dentro de la migración general:

- **Pagos:** nunca migres esto sin un ambiente de pruebas (sandbox) del proveedor. Corre transacciones de prueba en paralelo en ambos sistemas antes de mover tráfico real, y verifica que

los webhooks de confirmación de pago lleguen correctamente al sistema nuevo.

- **Email transaccional:** si usabas una librería específica del framework origen para mandar emails, migra a usar directamente la API del proveedor (SendGrid, Amazon SES, Postmark, etc.), que es igual de accesible desde cualquier lenguaje.
- **Colas y trabajos en segundo plano:** los jobs asíncronos (enviar un email, generar un reporte pesado, procesar un pago) necesitan su propio análisis de migración —identifica todos los "workers" del sistema legacy antes de dar la migración por completa, es un punto que se olvida seguido.

> **TIP**
>
> Usa el mismo skill de auditoría del capítulo 3 para generar un inventario específico de integraciones externas: busca en el código legacy todas las llamadas a APIs externas, credenciales de terceros, y jobs programados (cron). Es fácil que una integración vieja quede olvidada si no la buscas activamente.

Testing de la migración

CAPÍTULO 09

Por qué los tests importan más acá que en un proyecto nuevo

Cuando migras código, el riesgo más grande no es que el código nuevo no funcione —es que funcione distinto al original, de forma sutil, y que nadie lo note hasta que ya causó un problema en producción. Los tests son tu red de seguridad contra ese escenario.

El skill de testing

```
# skills/dotnet-testing/SKILL.md
---
name: dotnet-testing
description: Genera tests xUnit equivalentes a los tests originales del proyecto legacy
---
Pasos:
1. Leer los tests originales del módulo (PHPUnit u otro framework de origen)
2. Identificar casos de prueba y mocks usados
3. Generar los xUnit equivalentes:
   - Moq para mocks
   - FluentAssertions para los asserts
4. Generar tests de integración con WebApplicationFactory
5. Verificar que la cobertura sea al menos del 80%
```

Comparando comportamiento: sistema viejo vs. sistema nuevo, lado a lado

El paso final de validación: ejecuta la misma operación en ambos sistemas con los mismos datos de entrada, y compara los resultados directamente.

```
def comparar_respuestas(endpoint, datos):
    resp_legacy = llamar_api_legacy(endpoint, datos)
    resp_nueva = llamar_api_nueva(endpoint, datos)

    if resp_legacy.status != resp_nueva.status:
        print(f"Diferencia en status: {resp_legacy.status} vs {resp_nueva.status}")
        return False

    if resp_legacy.body != resp_nueva.body:
        print("Diferencia en el body (puede ser esperada, revisar a mano)")
        mostrar_diff(resp_legacy.body, resp_nueva.body)

    return True
```

TIP

No todas las diferencias son errores. A veces el sistema nuevo corrige un bug que la versión legacy tenía hace años, y el comportamiento "distinto" es en realidad el correcto. Documenta esos casos para que el equipo entienda por qué cambió, en vez de asumir automáticamente que es un error de la migración.

CI/CD para el proyecto nuevo

CAPÍTULO 10

Por qué esto no puede esperar hasta el final

Un error común es migrar varios módulos "a mano" y recién pensar en integración continua cuando el proyecto ya está grande. Al revés: configura CI/CD desde el primer módulo migrado, para que cada cambio nuevo se valide automáticamente antes de sumarse al resto, incluidos los cambios que generan los propios agentes de IA.

Pipeline de CI básico con GitHub Actions

```
# .github/workflows/ci.yml
name: CI

on:
  pull_request:
    branches: [main]
  push:
    branches: [main]

jobs:
  build-and-test:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

      - name: Configurar .NET
        uses: actions/setup-dotnet@v4
        with:
          dotnet-version: '8.0.x'

      - name: Restaurar dependencias
        run: dotnet restore

      - name: Compilar
        run: dotnet build --no-restore --configuration Release

      - name: Ejecutar tests
        run: dotnet test --no-build --configuration Release --logger "trx" --collect:"XPlat
Code Coverage"

      - name: Publicar cobertura
        uses: actions/upload-artifact@v4
        with:
          name: coverage-report
          path: '**/coverage.cobertura.xml'
```

Integrando la migración asistida por agentes al pipeline

Puedes automatizar incluso la parte de generar código con agentes dentro de un pipeline separado (no el de CI de cada commit, sino uno manual o programado), de forma que cada corrida de migración de un módulo quede documentada y sea repetible:

```
# .github/workflows/migracion-modulo.yml
name: Migrar modulo con agentes

on:
  workflow_dispatch:
    inputs:
      modulo:
        description: 'Nombre del modulo a migrar'
        required: true

jobs:
  migrar:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
      - name: Instalar OpenCode
        run: curl -fsSL https://opencode.ai/install.sh | sh
      - name: Ejecutar migracion del modulo
        run: |
          opencode run --agent migrator --skill dotnet-migration \
            --prompt "Migra el modulo ${{ github.event.inputs.modulo }} completo"
      - name: Build y test tras la migracion
        run: dotnet build && dotnet test
      - name: Crear Pull Request con los cambios
        uses: peter-evans/create-pull-request@v6
        with:
          title: "Migración automática: módulo ${{ github.event.inputs.modulo }}"
          branch: "migracion/${{ github.event.inputs.modulo }}"
```

> **NOTA**
>
> Nunca dejes que el código generado por un agente se mergee directo a la rama principal sin pasar por revisión humana. El patrón de arriba crea un Pull Request, no un merge automático —esa revisión manual es exactamente el punto donde tu criterio como ingeniero agrega el valor que el agente no puede.

Puesta en producción

CAPÍTULO 11

Estrategia Strangler Fig

No migres todo de una vez ni apagues la app original de un día para el otro. La estrategia Strangler Fig consiste en dejar la app original corriendo y redirigir el tráfico de a poco, módulo por módulo, hacia la nueva aplicación:

```
# nginx.conf
upstream app_legacy { server legacy-app:8080; }
upstream app_nueva { server nueva-app:5000; }

server {
    location /auth/ {
        proxy_pass http://app_nueva;   # Este módulo ya está migrado al stack nuevo
    }
    location / {
        proxy_pass http://app_legacy;  # El resto sigue en el stack original por ahora
    }
}
```

A medida que cada módulo termina su migración y pasa las pruebas, sumas su ruta al bloque del sistema nuevo. El día que todos los módulos estén migrados, apagas la app original con total tranquilidad —porque ya la venías reemplazando de a partes, con evidencia en cada paso.

Plan de rollback: qué hacer si algo sale mal

Ningún plan de migración está completo sin un plan de vuelta atrás. Antes de mover tráfico real a un módulo migrado, ten resuelto de antemano cómo revertir esa decisión en minutos, no en horas.

- **Rollback a nivel de proxy:** el cambio de ruta en el archivo de configuración del proxy (como el ejemplo de arriba) tiene que poder revertirse con un solo cambio y un reload, sin necesidad de un nuevo despliegue completo.
- **Compatibilidad de base de datos en ambos sentidos:** si el módulo nuevo escribe en la base de datos con un esquema distinto, asegúrate de que el sistema viejo pueda seguir leyendo esos datos (o de tener un mecanismo de sincronización) durante toda la transición, para que un rollback no pierda datos escritos por el sistema nuevo.
- **Feature flags para activar o desactivar el modulo nuevo:** en vez de depender solo del proxy, un flag a nivel de aplicación te permite apagar una funcionalidad migrada al instante si aparece un problema, sin tocar infraestructura.
- **Monitoreo con alertas desde el primer día:** definí de antemano qué métricas indican que algo salió mal (tasa de error, latencia, caída de conversiones) y a partir de qué umbral se dispara el

rollback, para no tener que decidirlo en caliente durante un incidente.

> **ATENCIÓN**
>
> El peor momento para diseñar tu plan de rollback es en medio de un incidente real. Escríbelo, pruébalo en un ambiente de pruebas, y asegúrate de que cualquier persona del equipo —no solo quien hizo la migración— sepa ejecutarlo.

Checklist final antes de dar la migración por terminada

- ☐ La app original sigue funcionando durante toda la transición?
- ☐ Todos los endpoints del sistema nuevo responden como se espera?
- ☐ Los tests xUnit pasan de forma consistente, integrados a CI/CD (capítulo 10)?
- ☐ La cobertura de tests es de al menos 80%?
- ☐ Las consultas a la base de datos se tradujeron correctamente?
- ☐ Los jobs y colas de tareas en segundo plano se migraron también (capítulo 8)?
- ☐ Los emails automáticos se siguen enviando igual que antes?
- ☐ Las integraciones con servicios externos siguen funcionando sin cambios visibles para el usuario?
- ☐ El módulo de autenticación fue migrado y probado con especial cuidado (capítulo 7)?
- ☐ Existe un plan de rollback probado para cada módulo antes de mover tráfico real?

Glosario y recursos

CAPÍTULO 12

Término	Qué significa
Clean Architecture	Forma de organizar el código en capas (Domain, Application, Infrastructure, Web) para que sea mantenible.
MediatR	Librería de .NET que implementa el patrón mediador, muy usada para separar Commands y Queries.
EF Core	Entity Framework Core, el ORM oficial de .NET para trabajar con bases de datos.
Dapper	Micro-ORM de .NET, más liviano que EF Core, ideal para consultas SQL de alto rendimiento.
Minimal API	Forma simplificada de definir endpoints HTTP en .NET, sin la estructura completa de un Controller.
Strangler Fig	Estrategia de migración gradual: reemplazar el sistema viejo de a partes mientras sigue funcionando.
xUnit	Framework de testing para .NET, equivalente a PHPUnit en el mundo PHP.
OpenCode	Herramienta que permite definir y ejecutar agentes de IA especializados sobre un repositorio de código.
JWT	JSON Web Token: un token firmado que lleva la información de sesión sin que el servidor tenga que guardarla.
Feature flag	Interruptor a nivel de código que permite activar o desactivar una funcionalidad sin un nuevo despliegue.
CI/CD	Integración y despliegue continuos: automatizar build, test y despliegue en cada cambio de código.
Rollback	Revertir un cambio o despliegue a su estado anterior cuando algo sale mal.

Para seguir aprendiendo

- La documentación oficial de tu framework destino (por ejemplo, .NET y Entity Framework Core).
- La documentación de tu framework de origen (Laravel, Symfony, etc.), para no perder detalles del comportamiento original.
- Los repositorios de ejemplo de Clean Architecture disponibles en GitHub, para tener una referencia de estructura.

. . .

Migrar una aplicación legacy con agentes de IA es un proceso que exige paciencia, método y las herramientas correctas. Con OpenCode, skills bien definidos, y un enfoque ordenado, puedes hacer en días lo que antes tomaba meses. Lo más importante para cerrar: no te saltees los pasos. Audita primero, diseña la arquitectura, migra de a módulos, testea cada uno, automatiza el pipeline y ten siempre un plan de vuelta atrás. La IA acelera muchísimo el proceso, pero el criterio y la revisión humana siguen siendo la parte que más cuida el resultado final.

Leandro Juan Sepúlveda Cuevas
Software Engineer · leandrosepulveda.com