IACADEMY

DE FUNDAMENTOS A ARQUITECTURA DE IA

MÓDULO 08

IA para desarrollo

Especialización

iacedemy.com — 2026

MÓDULO 08

IA para desarrollo

Nivel: Especialización

Autor: Ricardo Gutierrez

Publicación: Mayo 2026

Plataforma: iacedemy.com

TDD asistido por IA

La IA no reemplaza a los developers. Pero los developers que usan IA reemplazan a los que no la usan. La clave: TDD con IA no es "pide a la IA que escriba los tests". Es un proceso donde tu defines QUE debe pasar y la IA implementa COMO.

El ciclo TDD con IA

1. TU defines el comportamiento esperado (test spec)
2. IA escribe el test (tu revisas)
3. IA implementa el codigo para pasar el test
4. TU revisas que el test realmente valida lo que quieres

Demo: TDD para un endpoint API

```
 claude "Implementa TDD para un endpoint POST /api/v1/tickets.

Comportamiento esperado:
- Recibe JSON con: title (string, required), priority (enum: low/medium/high)
- Valida campos obligatorios, retorna 422 si faltan
- Crea el ticket en base de datos
- Retorna 201 con el ticket creado (incluye id y created_at)
- Si el usuario no esta autenticado, retorna 401

Paso 1: Escribe SOLO los tests (pytest + httpx).
No escribas la implementacion todavia."
```

Claude genera los tests:

```
 # tests/test_tickets.py
import pytest
from httpx import AsyncClient

@pytest.mark.asyncio
async def test_create_ticket_success(client: AsyncClient, auth_headers):
    response = await client.post("/api/v1/tickets", json={
        "title": "Server is down",
        "priority": "high"
    }, headers=auth_headers)
```

```
    assert response.status_code == 201
    data = response.json()
    assert "id" in data
    assert data["title"] == "Server is down"
    assert data["priority"] == "high"
    assert "created_at" in data

@pytest.mark.asyncio
async def test_create_ticket_missing_title(client: AsyncClient,
auth_headers):
    response = await client.post("/api/v1/tickets", json={
        "priority": "low"
    }, headers=auth_headers)
    assert response.status_code == 422

@pytest.mark.asyncio
async def test_create_ticket_invalid_priority(client: AsyncClient,
auth_headers):
    response = await client.post("/api/v1/tickets", json={
        "title": "Bug",
        "priority": "urgent"  # no es un valor valido
    }, headers=auth_headers)
    assert response.status_code == 422

@pytest.mark.asyncio
async def test_create_ticket_unauthenticated(client: AsyncClient):
    response = await client.post("/api/v1/tickets", json={
        "title": "Test",
        "priority": "low"
    })
    assert response.status_code == 401
```

Todos los tests fallan (rojo). Ahora pides la implementacion. Claude implementa el endpoint en FastAPI, ejecutas tests, todos pasan (verde).

Punto critico

Revisa los tests ANTES de pedir la implementacion. Si el test esta mal, la IA escribira codigo que pasa un test incorrecto. Basura entra, basura sale.

CLAUDE.md enterprise

Un CLAUDE.md basico dice "Este proyecto usa React y TypeScript". Un CLAUDE.md enterprise es un contrato entre tu equipo y la IA.

Estructura

```
 # CLAUDE.md — Ticketera SaaS

## 1. IDENTIDAD
SaaS B2B de gestion de tickets. Multi-tenant con RLS en Supabase.
Stack: FastAPI 0.111+ / Next.js 14 / Supabase PostgreSQL / Hetzner

## 2. RESTRICCIONES CRITICAS
- NUNCA queries sin filtro de tenant_id. SIEMPRE WHERE tenant_id = :tid
- NUNCA exponer stack traces al cliente. Loggear internamente.
- NUNCA usar ORM para queries complejas. SQL directo con parameterized
queries.
- Tests obligatorios para cada endpoint nuevo. Sin test = sin merge.

## 3. CONVENCIONES
- Endpoints: /api/v1/{recurso} (plural)
- Archivos: snake_case. Clases: PascalCase. Constantes: UPPER_SNAKE.
- Cada endpoint tiene: router, schema, service, test.

## 4. ANTI-PATTERNS
- NO usar try/except generico. Siempre excepciones especificas.
- NO crear funciones de mas de 30 lineas. Extraer.
- NO usar print(). Usar structlog.
- NO commitear .env. Verificar con pre-commit hook.
```

La seccion de RESTRICCIONES CRITICAS es la mas importante. Es donde le dices a la IA lo que NUNCA debe hacer. Sin ella, la IA tomara atajos que funcionan en el momento pero crean deuda tecnica.

Agentes especializados

Agente Arquitecto

```
 # .claude/commands/architect.md
Actua como arquitecto de software senior.

Analiza la peticion del usuario y:
1. Identifica los componentes afectados
2. Propone la arquitectura (diagrama ASCII)
3. Lista los archivos que hay que crear/modificar
4. Identifica riesgos y trade-offs
5. Propone un plan de implementacion paso a paso

NO escribas codigo. Solo el diseno.
Challengea la peticion si detectas problemas.
```

Ejemplo: "Necesito anadir un sistema de notificaciones push." Claude analiza el proyecto, propone arquitectura con event bus + worker + WebSocket, identifica 3 riesgos, sugiere plan de 4 pasos.

Agente Debugger

```
 # .claude/commands/debug.md
Actua como debugger senior.

Cuando el usuario reporta un bug:
1. Reproduce mentalmente el flujo
2. Identifica las 3 causas mas probables
3. Lee los archivos relevantes
4. Propone la causa raiz con evidencia
5. Sugiere el fix con diff exacto
6. Propone un test de regresion

NO apliques el fix automaticamente. Espera confirmacion.
```

Ejemplo: "El endpoint /api/v1/tickets devuelve 500 cuando el titulo tiene emojis." Claude lee el router, el schema, el service. Identifica que la columna de PostgreSQL tiene encoding incorrecto. Propone migracion + test de regresion.

Agente Code Reviewer

```
 # .claude/commands/review.md
Revisa el diff actual (git diff HEAD~1) y genera un code review.

Para cada finding:
- Severidad: CRITICAL / WARNING / INFO
- Archivo y linea
- Descripcion del problema
- Sugerencia de fix

Veredicto final: APPROVE / REQUEST_CHANGES / COMMENT
```

Code review en CI

```
 name: AI Code Review
on:
  pull_request:
    types: [opened, synchronize]

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
      - name: Get diff
        run: git diff origin/main...HEAD > diff.txt
      - name: AI Review
        run: python scripts/ai_review.py diff.txt > review.json
      - name: Post comments
        uses: actions/github-script@v7
        with:
          script: |
            const review = require('./review.json');
            for (const finding of review.findings) {
              await github.rest.pulls.createReviewComment({
                owner: context.repo.owner,
                repo: context.repo.repo,
                pull_number: context.issue.number,
```

```
                body: finding.severity + ': ' + finding.description,
                path: finding.file,
                line: finding.line,
                commit_id: context.payload.pull_request.head.sha
              });
            }
```

Quality gates

```
QUALITY_GATES = {
    "max_critical_findings": 0,    # 0 criticos para mergear
    "max_warning_findings": 3,     # maximo 3 warnings
    "min_test_coverage": 80,       # cobertura minima
    "max_function_complexity": 10, # complejidad ciclomatica
}
```

Si el PR tiene un finding CRITICAL, no se puede mergear. Automatico. Sin negociacion.

Cuando la IA ayuda vs cuando estorba

La IA AYUDA	La IA ESTORBA
Boilerplate (CRUD, schemas, tests)	Arquitectura critica (tu decides)
Refactoring mecanico	Logica de negocio compleja nueva
Documentacion de codigo existente	Documentacion de algo que no existe
Debugging con contexto (logs, stack trace)	Debugging sin contexto ("no funciona")
Code review automatizado	Decisiones de diseno (trade-offs)
Regex, SQL, config files	Algoritmos criticos de rendimiento

Regla de oro: usa IA para tareas donde un error se detecta rapido (tests fallan, lint falla). No la uses para decisiones donde un error tarda semanas en detectarse (arquitectura mala, modelo de datos incorrecto).

Ejercicio practico

Ejercicio M08: TDD + Agentes + CI

1. Elige un endpoint nuevo para tu proyecto.
2. Define el comportamiento esperado en lenguaje natural (5-7 puntos).
3. Pide a Claude que escriba SOLO los tests. Revisa antes de continuar.
4. Ejecuta tests (deben fallar). Pide la implementacion.
5. Crea un CLAUDE.md enterprise para tu proyecto (4 secciones minimo).
6. Configura los 3 agentes (architect, debug, review) como slash commands.
7. Ejecuta `/review` en tu ultimo commit.

Bonus: Configura un GitHub Action con AI Code Review en tu repositorio.

Conclusiones clave

Key takeaways del M08

1. TDD con IA: tu defines el QUE (comportamiento), la IA implementa el COMO (codigo).
2. CLAUDE.md enterprise tiene restricciones criticas, convenciones y anti-patterns. Es un contrato.
3. Tres agentes fundamentales: arquitecto (disena), debugger (diagnostica), reviewer (valida).
4. Code review automatizado en CI con quality gates bloquea PRs con findings criticos.
5. Usa IA donde los errores se detectan rapido. No la uses para decisiones estrategicas.
6. Revisa siempre los tests antes de pedir implementacion. Tests malos = codigo malo.

Tu workflow de desarrollo esta potenciado

En el M09 aplicamos todo esto a ciberseguridad: pipeline SOC con 4 agentes, GRC automatizado, y threat intelligence con correlacion.

Ir al Modulo 09

IACADEMY

iacedemy.com

De fundamentos a arquitectura de IA.
12 módulos prácticos. 24 recursos descargables.
Quizzes con certificado. Vídeos profesionales.

Empieza gratis en iacedemy.com/free