IACADEMY

DE FUNDAMENTOS A ARQUITECTURA DE IA

MÓDULO 14

Arquitectura de software con IA

Avanzado

iacedemy.com — 2026

MÓDULO 14

Arquitectura de software con IA

Nivel: Avanzado

Autor: Ricardo Gutierrez

Publicación: Mayo 2026

Plataforma: iacedemy.com

Clean architecture con IA

La IA puede generar codigo a velocidades que antes eran imposibles. Pero codigo rapido sin estructura es deuda tecnica rapida. Los principios de arquitectura limpia no cambian porque uses IA. De hecho, son mas importantes, porque la velocidad de generacion hace muy facil crear un monstruo inmantenible en pocas horas.

La separacion de responsabilidades sigue siendo la regla fundamental. Cada capa hace una cosa. Los detalles de implementacion (base de datos, framework web, API externa) nunca contaminan la logica de negocio.

Estructura de proyecto

```
proyecto/
 src/
   domain/              # Entidades y reglas de negocio
     models.py          # Modelos Pydantic puros (sin imports de framework)
     rules.py           # Validaciones de negocio
     exceptions.py      # Excepciones de dominio
   application/         # Casos de uso (orquesta domain)
     use_cases/
       create_task.py   # Un archivo por caso de uso
       assign_task.py
     services.py        # Servicios de aplicacion
     ports.py           # Interfaces abstractas (contratos)
   infrastructure/      # Implementaciones concretas
     database/
       supabase_repo.py # Implementacion de los ports con Supabase
       migrations/      # Archivos de migracion
     external/
       email_service.py # Implementacion de envio de emails
   api/                 # Capa de entrada (FastAPI)
     routes/
       tasks.py         # Endpoints de tareas
       users.py         # Endpoints de usuarios
     schemas.py         # Schemas de request/response
     dependencies.py    # Inyeccion de dependencias
 tests/
   unit/                # Tests de domain/ y application/
```

```
    integration/         # Tests de infrastructure/
    e2e/                 # Tests de api/ completos
```

La regla de dependencias: **las dependencias siempre apuntan hacia dentro**. La capa `api/` depende de `application/`, que depende de `domain/`. La capa `domain/` no importa nada de las otras capas. No sabe que existe FastAPI, ni Supabase, ni Cloudflare.

Prompt para que la IA respete la estructura

```
 Contexto: Proyecto FastAPI con clean architecture.
Estructura:
- domain/: modelos Pydantic puros y reglas de negocio. NO importa nada
externo.
- application/: use cases que orquestan domain/. Define ports (interfaces).
- infrastructure/: implementa los ports (Supabase, email, etc.)
- api/: endpoints FastAPI. Usa dependency injection para conectar capas.

Objetivo: Crear el flujo completo para "asignar una tarea a un usuario".

Restricciones:
- domain/ NO puede importar nada de infrastructure/ ni api/.
- Cada use case en su propio archivo.
- Tests unitarios para domain/ y application/ (sin base de datos real).
- Usa ABC para los ports en application/ports.py.
```

Cuando le das esta estructura a la IA, genera codigo limpio y bien organizado. Sin ella, genera todo en un solo archivo de 500 lineas.

Monolith first

Uno de los errores mas comunes con IA: como generar codigo es facil, los desarrolladores empiezan con microservicios, Docker Compose con 8 contenedores, message queues, API gateways y tres bases de datos. Para un proyecto con cero usuarios.

La regla: **empieza con un monolito. Extrae cuando algo duele.**

```
 # Para un MVP (0-1000 usuarios):

# CORRECTO
```

```
proyecto/
  backend/       # Un solo FastAPI
  frontend/      # Un solo Next.js
  database/      # Un solo PostgreSQL
  # Deploy: 1 servidor, 1 base de datos

# INCORRECTO (para un MVP)
proyecto/
  api-gateway/
  auth-service/
  task-service/
  notification-service/
  email-worker/
  redis/
  rabbitmq/
  postgres-auth/
  postgres-tasks/
  # Deploy: 9 contenedores, 2 bases de datos, 1 message queue
```

Senales de que necesitas extraer un servicio:

- Un componente consume 10x mas recursos que el resto (ej: procesamiento de imagenes)
- Un equipo necesita deployar independientemente sin afectar al resto
- Un componente tiene requisitos de escalado completamente diferentes
- Un componente falla y tumba todo lo demas

Si ninguna de estas se cumple, el monolito es la respuesta correcta. Es mas simple de desarrollar, deployar, debuggear y monitorizar.

Diseno de APIs REST

Para el 90% de los proyectos, REST es suficiente. GraphQL resuelve un problema especifico: cuando tienes multiples clientes con necesidades de datos muy diferentes (app movil que quiere pocos campos, dashboard web que quiere muchos). Si tienes un solo frontend, REST es mas simple y predecible.

API REST con buenas practicas en FastAPI

```
 from fastapi import FastAPI, HTTPException, Depends, Query
from pydantic import BaseModel, Field
from datetime import datetime
from enum import Enum

app = FastAPI(
    title="Task Management API",
    version="1.0.0",
    docs_url="/api/docs",
    redoc_url="/api/redoc"
)

# --- Schemas (contratos claros) ---

class TaskStatus(str, Enum):
    pending = "pending"
    in_progress = "in_progress"
    done = "done"

class TaskCreate(BaseModel):
    title: str = Field(..., min_length=1, max_length=200,
                       description="Titulo de la tarea")
    description: str | None = Field(None, max_length=5000)
    priority: int = Field(default=1, ge=1, le=5,
                          description="1=baja, 5=critica")
    assigned_to: str | None = None

class TaskUpdate(BaseModel):
    title: str | None = Field(None, min_length=1, max_length=200)
    status: TaskStatus | None = None
    priority: int | None = Field(None, ge=1, le=5)

class TaskResponse(BaseModel):
    id: int
    title: str
    description: str | None
    priority: int
    status: TaskStatus
    assigned_to: str | None
    created_at: datetime
    updated_at: datetime | None
```

```
class PaginatedResponse(BaseModel):
    data: list[TaskResponse]
    total: int
    page: int
    per_page: int

# --- Endpoints (versionados) ---

@app.get("/api/v1/tasks", response_model=PaginatedResponse)
async def list_tasks(
    status: TaskStatus | None = None,
    priority: int | None = Query(None, ge=1, le=5),
    page: int = Query(1, ge=1),
    per_page: int = Query(20, ge=1, le=100),
    user=Depends(get_current_user)
):
    """Listar tareas del usuario con filtros opcionales."""
    tasks, total = await task_service.list(
        user_id=user.id, status=status,
        priority=priority, page=page, per_page=per_page
    )
    return PaginatedResponse(
        data=tasks, total=total, page=page, per_page=per_page
    )

@app.post("/api/v1/tasks", response_model=TaskResponse, status_code=201)
async def create_task(task: TaskCreate, user=Depends(get_current_user)):
    """Crear una nueva tarea."""
    try:
        return await task_service.create(task, user.id)
    except DuplicateError:
        raise HTTPException(409, "Ya existe una tarea con ese titulo")

@app.patch("/api/v1/tasks/{task_id}", response_model=TaskResponse)
async def update_task(
    task_id: int, update: TaskUpdate, user=Depends(get_current_user)
):
    """Actualizar campos de una tarea existente."""
    task = await task_service.get(task_id, user.id)
    if not task:
        raise HTTPException(404, "Tarea no encontrada")
    return await task_service.update(task_id, update, user.id)

@app.delete("/api/v1/tasks/{task_id}", status_code=204)
async def delete_task(task_id: int, user=Depends(get_current_user)):
```

```
    """Eliminar una tarea (soft delete)."""
    deleted = await task_service.delete(task_id, user.id)
    if not deleted:
        raise HTTPException(404, "Tarea no encontrada")
```

Checklist de API

1. **Versionado en la URL**: `/api/v1/`. Cuando hagas breaking changes, creas `/api/v2/`.
2. **Validacion con Pydantic**: tipos, limites, formatos. FastAPI valida automaticamente.
3. **Codigos HTTP correctos**: 201 para creacion, 204 para eliminacion, 404 para no encontrado, 409 para conflicto, 422 para validacion.
4. **Documentacion automatica**: FastAPI genera `/docs` (Swagger) y `/redoc` gratis.
5. **Paginacion**: nunca devuelvas listas sin paginar. Parametros `page` y `per_page` con limites.
6. **Autenticacion**: en cada endpoint que lo necesite, via `Depends()`.

Patrones con IA

Cuatro patrones que funcionan especialmente bien cuando usas IA para desarrollar:

1. Coordinator + Workers

Ya lo viste en M04. Un agente planifica, N ejecutan. Funciona para refactors, migraciones, auditorias. La clave: el Coordinator lee los resultados reales de cada Worker, nunca delega a ciegas.

2. Event-driven

Algo pasa (nuevo usuario, pedido completado) y se disparan acciones asincronas. Perfecto para n8n, webhooks, notificaciones.

```
 # Evento: nuevo usuario registrado
async def on_user_registered(user: User):
    # Acciones independientes, se ejecutan en paralelo
    await asyncio.gather(
        send_welcome_email(user),
        create_default_project(user),
```

```
        notify_slack_channel(f"Nuevo usuario: {user.email}"),
        track_event("user_registered", user.id)
    )
```

3. CQRS simplificado

Separa las operaciones de lectura (queries) de las de escritura (commands). No necesitas Event Sourcing completo. Solo separar endpoints y modelos de datos optimizados para cada operacion.

```
 # Command: optimizado para escritura (validacion completa)
class CreateTaskCommand(BaseModel):
    title: str = Field(..., min_length=1, max_length=200)
    description: str | None = None
    priority: int = Field(default=1, ge=1, le=5)

# Query: optimizado para lectura (campos que necesita el frontend)
class TaskListView(BaseModel):
    id: int
    title: str
    priority: int
    status: str
    assignee_name: str | None  # JOIN pre-computado
    days_open: int             # Calculado en la query
```

4. Repository pattern

Abstrae el acceso a datos. Tu use case llama a `repo.get_by_id(id)`, no a SQL directo. Facilita cambiar de base de datos, mockear en tests, y mantener la logica de negocio limpia.

```
 # application/ports.py (interfaz)
from abc import ABC, abstractmethod

class TaskRepository(ABC):
    @abstractmethod
    async def get_by_id(self, task_id: int, user_id: str) -> Task | None: ...

    @abstractmethod
    async def save(self, task: Task) -> Task: ...
```

```
    @abstractmethod
    async def list_by_user(self, user_id: str, filters: dict) ->
list[Task]: ...

# infrastructure/supabase_repo.py (implementacion)
class SupabaseTaskRepository(TaskRepository):
    def __init__(self, client):
        self.client = client

    async def get_by_id(self, task_id: int, user_id: str) -> Task | None:
        result = await self.client.table("tasks") \
            .select("*") \
            .eq("id", task_id) \
            .eq("user_id", user_id) \
            .single() \
            .execute()
        return Task(**result.data) if result.data else None

# tests/unit/test_create_task.py (test con mock)
class FakeTaskRepository(TaskRepository):
    def __init__(self):
        self.tasks = []

    async def save(self, task: Task) -> Task:
        task.id = len(self.tasks) + 1
        self.tasks.append(task)
        return task
```

Refactoring asistido por IA

La IA es excelente identificando code smells y sugiriendo mejoras. Pero hay una regla: **mide antes, refactoriza, mide despues**. Sin metricas, no sabes si mejoraste algo.

```
 # Paso 1: medir
# Complejidad ciclomatica con radon
pip install radon
radon cc src/ -s -n C  # Solo muestra funciones con complejidad >= C

# Cobertura de tests
pytest --cov=src --cov-report=term-missing

# Paso 2: pedirle a la IA que identifique problemas
```

```
# Prompt:
"Analiza src/services/task_service.py. Identifica:
1. Funciones con mas de 20 lineas
2. Codigo duplicado
3. Responsabilidades mezcladas (una funcion que hace varias cosas)
4. Manejo de errores ausente o generico
No arregles nada todavia. Solo lista los problemas."

# Paso 3: refactorizar con objetivo claro
"Refactoriza la funcion process_task() de src/services/task_service.py.
Objetivo: reducir complejidad ciclomatica de 15 a menos de 5.
Metodo: extraer funciones helper con nombres descriptivos.
Restriccion: no cambiar la firma ni el comportamiento externo."

# Paso 4: medir de nuevo
radon cc src/ -s -n C
pytest --cov=src --cov-report=term-missing
```

Arquitectura de testing

La piramide de testing no cambia con IA. Lo que cambia es que la IA puede generar tests mucho mas rapido. Pero necesitas saber que tests pedir.

```
 # Piramide de tests:
# Muchos unitarios (domain/ y application/): rapidos, sin dependencias
# Algunos de integracion (infrastructure/): con DB real o mock
# Pocos e2e (api/): endpoints completos, lentos

# Test unitario (domain/)
def test_task_cannot_have_priority_zero():
    with pytest.raises(ValidationError):
        Task(title="Test", priority=0)

def test_task_title_max_length():
    with pytest.raises(ValidationError):
        Task(title="x" * 201, priority=1)

# Test de integracion (infrastructure/)
async def test_supabase_repo_saves_and_retrieves():
    repo = SupabaseTaskRepository(test_client)
    task = Task(title="Integration test", priority=3, user_id="test-user")
    saved = await repo.save(task)
```

```
    retrieved = await repo.get_by_id(saved.id, "test-user")
    assert retrieved.title == "Integration test"

# Test e2e (api/)
async def test_create_task_endpoint():
    response = await client.post("/api/v1/tasks", json={
        "title": "E2E test task",
        "priority": 4
    }, headers={"Authorization": f"Bearer {token}"})
    assert response.status_code == 201
    assert response.json()["title"] == "E2E test task"
```

Prompt para generar tests

```
 "Genera tests para src/application/use_cases/create_task.py.
Incluye:
- Happy path: crear tarea con datos validos
- Titulo vacio: debe lanzar ValidationError
- Prioridad fuera de rango: debe lanzar ValidationError
- Usuario no existe: debe lanzar NotFoundError
- Tarea duplicada: debe lanzar DuplicateError
Usa pytest + pytest-asyncio. Mock el repository con la clase
FakeTaskRepository."
```

ADRs: documentar decisiones

Cuando la IA sugiere una decision de arquitectura (framework, patron, libreria), documenta el por que. Un ADR (Architecture Decision Record) es un documento de 1 pagina con 4 secciones.

```
 # docs/adr/ADR-001-fastapi-backend.md

# ADR-001: FastAPI como framework backend

## Estado
Aceptado - 2026-05-12

## Contexto
Necesitamos un framework Python para el backend de la aplicacion.
Requisitos: async nativo, validacion automatica, documentacion OpenAPI,
```

```
rendimiento alto, ecosistema maduro.

## Decision
Usamos FastAPI porque:
1. Async nativo con ASGI (importante para I/O con Supabase y APIs externas)
2. Validacion automatica con Pydantic (reduce bugs y documentacion manual)
3. Documentacion OpenAPI generada automaticamente (/docs)
4. Rendimiento comparable a Node.js/Go para I/O bound
5. Ecosistema Python: acceso a librerias ML/AI directamente

Alternativas descartadas:
- Django: demasiado opinionado, ORM no necesario con Supabase
- Flask: sin async nativo, menos estructura
- Express.js: cambiaria el lenguaje del stack sin beneficio claro

## Consecuencias
- Positivo: desarrollo rapido, documentacion gratis, tipado fuerte
- Negativo: menos talent pool que Django, requiere entender async/await
- Riesgo: si el equipo crece mucho, Django tiene mas estructura "por
defecto"
```

Sin ADRs, cuando algo falla 6 meses despues, nadie recuerda por que se tomo esa decision. Con ADRs, abres el archivo y tienes el contexto completo.

Diseno de base de datos con IA

La IA es excelente generando esquemas de base de datos. Pero necesitas darle contexto sobre tu dominio, relaciones y restricciones.

```
 # Prompt para generar esquema
"Disena un esquema PostgreSQL para una aplicacion de gestion de proyectos.

Entidades:
- Organization: nombre, plan (free/pro/enterprise), created_at
- User: email, name, role (admin/member/viewer), organization_id
- Project: name, description, status, organization_id
- Task: title, description, priority (1-5), status, project_id, assigned_to

Requisitos:
- Multi-tenant: cada organizacion solo ve sus datos (RLS)
- Soft delete: campo deleted_at nullable en todas las tablas
- Timestamps: created_at y updated_at automaticos
```

```
- Indices para las queries mas comunes (listar tareas por proyecto, por
usuario)
- Foreign keys con ON DELETE CASCADE donde tenga sentido

Genera: CREATE TABLE, CREATE INDEX, ALTER TABLE para RLS, politicas RLS."
```

La IA generara un esquema completo. Tu trabajo: **revisar cada foreign key, cada indice, cada politica RLS**. Verificar que las relaciones tienen sentido, que los tipos son correctos, y que las politicas de seguridad no tienen huecos.

Anti-patterns al usar IA para arquitectura

Estos errores los veo todas las semanas. Aprende a detectarlos:

1. **Over-engineering**: la IA sugiere 3 capas de abstraccion, Strategy pattern, Factory pattern y Observer pattern para un CRUD. Si la solucion mas simple funciona, usa la solucion mas simple.
2. **Aceptar sin entender**: copiar codigo generado por la IA sin leer cada linea. Si no entiendes que hace, no lo pongas en produccion.
3. **Sin ADRs**: la IA toma decisiones (framework, libreria, patron) y nadie documenta por que. Cuando algo falla, no sabes la razon de cada decision.
4. **Sin tests**: "la IA lo genero, funcionara". No. La piramide de testing sigue siendo: muchos unitarios, algunos de integracion, pocos e2e.
5. **Refactoring sin metricas**: pedirle a la IA que refactorice sin saber que mejorar. Mide complejidad antes, refactoriza, mide despues.
6. **Microservicios prematuros**: como generar codigo es rapido, empezar con 8 servicios para un MVP. Monolith first.
7. **Cargo cult architecture**: copiar la arquitectura de Netflix porque "ellos lo hacen asi". Tu proyecto no tiene 200 millones de usuarios.

> **Regla anti over-engineering**
>
> Antes de aceptar una sugerencia de la IA, pregunta: "Puedo resolver esto con una funcion de 10 lineas?" Si la respuesta es si, la abstraccion compleja es innecesaria.

Ejercicio practico

Ejercicio M14: Arquitectura de un proyecto real

1. **Estructura**: Crea un proyecto con la estructura clean architecture (domain/, application/, infrastructure/, api/). Usa el prompt de ejemplo para que la IA genere el scaffolding.
2. **API**: Implementa 3 endpoints CRUD con FastAPI siguiendo las buenas practicas: versionado, validacion Pydantic, codigos HTTP correctos, paginacion.
3. **Repository pattern**: Implementa un port (interfaz) y dos implementaciones: una con Supabase para produccion y una FakeRepository para tests.
4. **Tests**: Escribe al menos 5 tests unitarios para domain/ y 3 para application/ usando el FakeRepository.
5. **ADRs**: Crea 3 ADRs documentando las decisiones principales de tu proyecto (framework, base de datos, patron de arquitectura).
6. **Metricas**: Ejecuta radon para medir complejidad ciclomatica. Si alguna funcion tiene complejidad > 5, pidele a la IA que la refactorice.

Bonus: Configura un hook PreCommit que ejecute ruff + pytest antes de cada commit.

Conclusiones clave

Key takeaways del M14

1. Clean architecture sigue siendo obligatoria con IA. Dependencias hacia dentro, domain/ no importa nada externo.
2. Monolith first. Microservicios solo cuando algo especifico duele (recursos, equipos, escalado).
3. APIs REST: versionado, validacion Pydantic, codigos HTTP, paginacion, documentacion auto.

4. Repository pattern abstrae datos. CQRS separa reads/writes. Event-driven para acciones asincronas.
5. Refactoring con metricas: mide antes, refactoriza, mide despues. Sin datos no hay mejora real.
6. ADRs documentan decisiones. Sin ellos, en 6 meses nadie sabe por que se eligio cada tecnologia.
7. Anti-patterns: over-engineering, aceptar sin entender, microservicios prematuros, refactoring sin metricas.

Siguiente: M15 - Bases de datos e IA

Ahora que tienes la arquitectura clara, vamos a profundizar en bases de datos: SQL practico, PostgreSQL, Supabase, RLS, migraciones y optimizacion con IA.

Ir al Modulo 15

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