

MÓDULO 8

Refactorización, revisión de código y optimización con IA

[/review](#) · [Code Review](#) · [Refactor](#) · [Baselines](#) · [Perfiles](#)

Plan de Formación — OpenAI Codex para Desarrolladores

Nivel: Intermedio-Avanzado · Duración estimada: 3–4 horas

Versión 1.0 — Febrero 2026

Índice

Índice.....	2
1. Introducción.....	4
2. Objetivos de aprendizaje	4
3. Contenidos teóricos	5
3.1 /review — El Revisor Integrado de Codex.....	5
3.1.1 Modos de revisión.....	5
3.1.2 Modelo de review.....	5
3.2 Code Review en GitHub.....	5
3.3 Prompt de Review Estructurado	5
3.4 Refactorización Incremental.....	6
3.4.1 Strangler Pattern.....	6
3.4.2 Snapshot de Tests (Green Baseline).....	6
3.4.3 Small PRs + Safe Steps	6
3.4.4 Invariante: NO BEHAVIOUR CHANGE.....	6
3.5 Optimización con Baselines	6
3.6 Perfiles para Review vs. Edit	6
3.7 Codex GitHub Action para Code Review.....	7
4. Buenas prácticas	8
4.1 Review en formato PR	8
4.2 Tests antes de tocar lógica	8
4.3 Perfil read-only para reviews.....	8
4.4 Incremental, no big-bang	8
4.5 Baseline antes de optimizar	8
4.6 AGENTS.md con Review Guidelines.....	8
5. Errores comunes y cómo evitarlos	9
5.1 Refactor sin tests.....	9
5.2 Optimización prematura sin baseline.....	9
5.3 Big-bang refactor.....	9
5.4 Ignorar hallazgos de /review	9
5.5 Review con perfil edit	9
6. Casos de uso reales	10
6.1 Reducir duplicación en módulo legacy	10
6.2 Review de PR para seguridad	10
6.3 Optimizar hot path.....	10
6.4 Codex GitHub Action para reviews automatizadas	10
7. Laboratorio práctico (L8) — Codex como Revisor de PR	11

7.1 Paso 1 — Crear proyecto base.....	11
7.2 Paso 2 — Crear rama con cambios imperfectos	12
7.3 Paso 3 — Revisar con /review	13
7.4 Paso 4 — Aplicar correcciones con Codex	14
7.5 Paso 5 — Re-review para confirmar	14
7.6 Paso 6 — Verificar resultado	14
7.7 Limpieza (Protocolo Obligatorio).....	15
8. Resumen y siguientes pasos	16
8.1 Conceptos clave.....	16
8.2 Preparación para el Módulo 9	16
9. Referencias y recursos	16

1. Introducción

Hasta ahora hemos usado Codex para escribir código nuevo. Pero en la realidad profesional, la mayor parte del tiempo se dedica a revisar, refactorizar y optimizar código existente. Codex incluye herramientas específicas para estas tareas: el comando `/review` para revisiones priorizadas, perfiles `read-only` para análisis sin riesgo, y un flujo disciplinado de refactorización incremental con baselines medibles.

En este módulo aprenderemos a usar Codex como revisor de código, refactorizador seguro y optimizador con evidencia. El flujo clave es: medir → revisar → corregir → verificar → medir de nuevo.

2. Objetivos de aprendizaje

#	Objetivo	Evidencia de logro
O1	Usar <code>/review</code> de Codex CLI para obtener revisiones priorizadas.	Hallazgos P0/P1 identificados con fichero y líneas.
O2	Aplicar refactorización incremental con invariante de tests.	Green baseline → refactor → green post-refactor.
O3	Medir optimizaciones con baselines explícitos.	Métrica antes vs. después documentada.
O4	Diferenciar perfiles <code>review-only</code> vs. <code>dev-edit</code> .	Perfil <code>read-only</code> impide modificaciones.

3. Contenidos teóricos

3.1 /review — El Revisor Integrado de Codex

Codex CLI incluye el comando `/review` que lanza un agente reviewer dedicado. Este agente lee el diff seleccionado y reporta hallazgos priorizados y accionables sin tocar el working tree.

3.1.1 Modos de revisión

Modo	Descripción
Review against a base branch	Selecciona rama local; Codex calcula merge base y revisa diff.
Review uncommitted changes	Revisa staged, unstaged y untracked.
Review a commit	Lista commits recientes y revisa el SHA seleccionado.
Custom review instructions	Prompt personalizado (ej. "Focus on security").

```
/review
```

```
# Codex muestra los modos disponibles y pide seleccionar.
# El resultado aparece como un turno independiente en la transcripción.
# Usar /diff después para inspeccionar los cambios exactos.
```

3.1.2 Modelo de review

Por defecto, `/review` usa el modelo de la sesión actual. Se puede configurar un modelo específico:

```
# ~/.codex/config.toml
review_model = "gpt-5.3-codex"
```

3.2 Code Review en GitHub

Codex puede revisar PRs directamente en GitHub de dos formas:

- Automático: Habilitar Codex Code Review en el repo. Codex revisa automáticamente al abrir PRs.
- Manual: Escribir `@codex review` en un comentario de PR. Opcionalmente añadir foco: `@codex review for security vulnerabilities`.

Prioridades: En GitHub, Codex solo reporta issues **P0** (bloqueantes) y **P1** (importantes). Para issues menores, configurar en AGENTS.md: `Treat documentation typos as P1.`

i AGENTS.md con Review Guidelines

Codex busca ficheros AGENTS.md en el repo y sigue las secciones de "Review guidelines" para personalizar qué revisa y con qué prioridad.

3.3 Prompt de Review Estructurado

GPT-5.2-Codex ha recibido entrenamiento específico para code review. El prompt recomendado por OpenAI incluye:

- Foco en: correctness, performance, security, maintainability, developer experience.
- Solo issues accionables introducidos por el PR.
- Cada issue: explicación directa + fichero y rango de líneas exactos.
- Priorizar issues severos; evitar nit-level salvo que bloqueen comprensión.
- Veredicto final: "patch is correct" o "patch is incorrect" con justificación y confidence score (0–1).

3.4 Refactorización Incremental

Técnicas para refactorizar de forma segura con Codex:

3.4.1 Strangler Pattern

Reemplazar componentes legacy progresivamente, manteniendo la funcionalidad antigua activa hasta que la nueva esté verificada con tests.

3.4.2 Snapshot de Tests (Green Baseline)

Ejecutar tests ANTES del refactor. Si alguno falla antes de empezar, es un problema pre-existente, no una regresión del refactor.

3.4.3 Small PRs + Safe Steps

Cada refactor como commit atómico. Si un paso rompe algo, revert limpio sin afectar otros cambios.

3.4.4 Invariante: NO BEHAVIOUR CHANGE

Regla de oro

Los tests existentes deben pasar sin modificación después de cada paso de refactorización. Si necesitas cambiar los tests, ese es un paso separado que requiere justificación.

3.5 Optimización con Baselines

Antes de optimizar, siempre medir:

- 1. Establecer baseline: Medir la métrica actual (latencia, memoria, complejidad ciclomática).
- 2. Pedir a Codex la optimización con el baseline como referencia explícita en el prompt.
- 3. Medir después: Comparar con el baseline.
- 4. Aceptar solo si mejora y los tests siguen pasando.

Optimización prematura

No pidas "hazlo más rápido" sin medir primero. Incluye el baseline en el prompt: "La latencia actual es 340ms. Optimiza para reducirla bajo 200ms sin romper tests."

3.6 Perfiles para Review vs. Edit

Usar perfiles del Módulo 6 para separar los roles de revisión y edición:

```
[profiles.review-only]
sandbox_mode = "read-only"
approval_policy = "untrusted"
review_model = "gpt-5.3-codex"
model_reasoning_effort = "high"
# Codex NO puede modificar ficheros

[profiles.dev-edit]
sandbox_mode = "workspace-write"
approval_policy = "on-request"
# Codex puede editar tras aprobación
```

```
# Revisar sin riesgo de modificar:
codex --profile review-only

# Editar tras la revisión:
codex --profile dev-edit
```

3.7 Codex GitHub Action para Code Review

Para automatizar reviews en CI/CD, usar openai/codex-action@v1 en GitHub Actions:

- Instala Codex CLI automáticamente en el runner.
- Ejecuta codex exec con un prompt-file de review.
- Postea hallazgos como comentario en la PR.
- safety-strategy: drop-sudo elimina sudo antes de ejecutar (protege secretos).

4. Buenas prácticas

4.1 Review en formato PR

Pedir a Codex que reporte riesgos, edge cases y tests faltantes con fichero y línea. Usar /review para revisiones estructuradas.

4.2 Tests antes de tocar lógica

Exigir que Codex ejecute tests primero (green baseline) antes de cualquier refactor. Si algún test falla antes del refactor, es un problema pre-existente.

4.3 Perfil read-only para reviews

Usar el perfil review-only (read-only + untrusted) para que Codex analice sin riesgo de modificar código inadvertidamente.

4.4 Incremental, no big-bang

Cada refactor como commit atómico. Usar /review después de cada paso para verificar que no se introdujeron regresiones.

4.5 Baseline antes de optimizar

Siempre medir antes y después. No aceptar "optimizaciones" sin evidencia medible.

4.6 AGENTS.md con Review Guidelines

Definir qué debe revisar Codex (seguridad, estilo, tests faltantes) para que las revisiones sean consistentes entre sesiones y entre miembros del equipo.

5. Errores comunes y cómo evitarlos

5.1 Refactor sin tests

Problema: Modificar lógica sin green baseline. Regresiones invisibles.

Solución: Siempre ejecutar tests antes y después. El invariante es: los tests existentes pasan sin modificación.

5.2 Optimización prematura sin baseline

Problema: "Hazlo más rápido" sin medir qué tan rápido es ahora.

Solución: Incluir baseline medible en el prompt. Medir después y comparar.

5.3 Big-bang refactor

Problema: Reescribir un módulo completo en un solo commit.

Solución: Strangler pattern, safe steps, commits atómicos.

5.4 Ignorar hallazgos de /review

Problema: Pedir revisión pero no actuar sobre los findings.

Solución: Flujo disciplinado: /review → fix → /review para confirmar que los issues se resolvieron.

5.5 Review con perfil edit

Problema: Usar --full-auto durante review. Codex podría modificar código sin supervisión.

Solución: Perfil review-only (read-only + untrusted). Cambiar a dev-edit solo cuando se van a aplicar correcciones.

6. Casos de uso reales

6.1 Reducir duplicación en módulo legacy

Usar Codex para identificar código duplicado en un módulo grande, extraer funciones comunes y verificar con tests que el comportamiento no cambió. Codex ejecuta tests antes/después y muestra git diff --stat.

6.2 Review de PR para seguridad

En GitHub: @codex review for security. En local: /review con custom instructions "Focus on SQL injection, XSS, and secrets in code".

6.3 Optimizar hot path

Medir latencia con time o cProfile. Dar el baseline a Codex en el prompt. Pedir optimización específica. Medir de nuevo y aceptar solo si mejora.

6.4 Codex GitHub Action para reviews automatizadas

Configurar openai/codex-action@v1 en el pipeline de CI para que revise automáticamente cada PR. Los hallazgos se postean como comentarios en la PR, priorizados por severidad.

7. Laboratorio práctico (L8) — Codex como Revisor de PR

⌚ Objetivo del laboratorio

Crear una rama con código deliberadamente imperfecto (SQL injection, falta de validación), usar /review para obtener hallazgos priorizados, aplicar correcciones y verificar con re-review.

7.1 Paso 1 — Crear proyecto base

Tiempo estimado: 5 minutos

```
mkdir /tmp/codex-lab08 && cd /tmp/codex-lab08
git init

python3 -m venv .venv
source .venv/bin/activate
pip install pytest flake8

cat > .gitignore << 'EOF'
.venv/
__pycache__/
*.pyc
EOF

mkdir -p src tests

cat > src/auth.py << 'PYEOF'
"""Módulo de autenticación (versión base, correcta)."""
import hashlib
import os

def hash_password(password: str) -> str:
    salt = os.urandom(16).hex()
    hashed = hashlib.sha256((salt + password).encode()).hexdigest()
    return f"{salt}:{hashed}"

def verify_password(password: str, stored: str) -> bool:
    salt, hashed = stored.split(":")
    check = hashlib.sha256((salt + password).encode()).hexdigest()
    return check == hashed
PYEOF

cat > tests/test_auth.py << 'PYEOF'
from src.auth import hash_password, verify_password

def test_hash_and_verify():
    h = hash_password("secret123")
    assert verify_password("secret123", h)

def test_wrong_password():
    h = hash_password("secret123")
    assert not verify_password("wrong", h)
```

```

def test_different_hashes():
    h1 = hash_password("same")
    h2 = hash_password("same")
    assert h1 != h2
PYEOF

PYTHONPATH=. pytest tests/ -v
git add -A && git commit -m "chore: módulo auth base"

```

7.2 Paso 2 — Crear rama con cambios imperfectos

Tiempo estimado: 5 minutos

```

git checkout -b feature/user-management

cat > src/users.py << 'PYEOF'
"""Gestión de usuarios – código deliberadamente imperfecto."""
import sqlite3
from src.auth import hash_password

DB_PATH = "users.db"

def init_db():
    conn = sqlite3.connect(DB_PATH)
    conn.execute("""
        CREATE TABLE IF NOT EXISTS users (
            id INTEGER PRIMARY KEY,
            username TEXT,
            password TEXT,
            role TEXT DEFAULT 'user'
        )
    """)
    conn.commit()
    conn.close()

def create_user(username, password, role="user"):
    # BUG 1: No valida inputs
    # BUG 2: SQL injection en username
    conn = sqlite3.connect(DB_PATH)
    hashed = hash_password(password)
    conn.execute(
        f"INSERT INTO users (username, password, role) "
        f"VALUES ('{username}', '{hashed}', '{role}')"
    )
    conn.commit()
    conn.close()

def get_user(username):
    # BUG 3: SQL injection
    conn = sqlite3.connect(DB_PATH)
    cursor = conn.execute(
        f"SELECT * FROM users WHERE username = '{username}'"
    )
    row = cursor.fetchone()

```

```

conn.close()
if row:
    return {"id": row[0], "username": row[1],
            "password": row[2], "role": row[3]}
return None

def delete_user(username):
    # BUG 4: No verifica existencia
    # BUG 5: Devuelve siempre True
    conn = sqlite3.connect(DB_PATH)
    conn.execute(
        f"DELETE FROM users WHERE username = '{username}'"
    )
    conn.commit()
    conn.close()
    return True

def is_admin(username):
    user = get_user(username)
    # BUG 6: Sin manejo de None
    return user["role"] == "admin"
PYEOF

cat > tests/test_users.py << 'PYEOF'
import os
import pytest
from src.users import init_db, create_user, get_user, DB_PATH

@pytest.fixture(autouse=True)
def clean_db():
    yield
    if os.path.exists(DB_PATH):
        os.remove(DB_PATH)

def test_create_and_get():
    init_db()
    create_user("alice", "pass123")
    user = get_user("alice")
    assert user is not None
    assert user["username"] == "alice"
PYEOF

PYTHONPATH=. pytest tests/ -v
git add -A && git commit -m "feat: user management (WIP)"
```

7.3 Paso 3 — Revisar con /review

Tiempo estimado: 10 minutos

```
codex --profile review-only
# (o: codex → /permissions → Read Only)
```

Escribir en el TUI:

```
/review

# Seleccionar: "Review against a base branch" → main
# Codex analiza diff main..feature/user-management
```

Hallazgos esperados:

- P0/P1: SQL injection en `create_user`, `get_user`, `delete_user` (f-strings con input de usuario).
- P1: No validación de inputs en `create_user` (username vacío, role arbitrario).
- P1: `is_admin` sin manejo de None (KeyError si usuario no existe).
- P2: `delete_user` no verifica existencia, devuelve siempre True.

7.4 Paso 4 — Aplicar correcciones con Codex*Tiempo estimado: 10 minutos*

```
codex --full-auto
```

Prompt en el TUI:

Aplica las correcciones de seguridad:

1. Corregir SQL injection en `@src/users.py` usando parámetros `sqlite3` (?) en lugar de f-strings.
2. Validar inputs en `create_user` (username no vacío, role en `['user', 'admin']`).
3. Manejar None en `is_admin`.
4. `delete_user` debe verificar si el usuario existe.
5. Añadir tests en `@tests/test_users.py` para los edge cases.

Ejecuta `pytest` y `flake8` al terminar.

7.5 Paso 5 — Re-review para confirmar*Tiempo estimado: 5 minutos*

```
codex

# En el TUI:
/review
# Seleccionar: "Review uncommitted changes"
#   o "Review against a base branch" → main

# Los hallazgos P0/P1 deben haber desaparecido.
```

7.6 Paso 6 — Verificar resultado

```
PYTHONPATH=.. pytest tests/ -v
```

```
flake8 src/ tests/
git diff --stat
git add -A && git commit -m "fix: SQL injection + input validation"
```

Verificación	Resultado esperado
/review encontró bugs	P0: SQL injection, P1: validación, P1: None handling
SQL injection corregido	Parámetros (?) en lugar de f-strings
Validación añadida	username no vacío, role validado
is_admin maneja None	No lanza KeyError
Tests nuevos	Edge cases cubiertos
Tests pasan	pytest verde
Lint limpio	flake8 sin errores

7.7 Limpieza (Protocolo Obligatorio)

⚠️ Limpieza de recursos

Ejecutar al finalizar:

```
deactivate
rm -rf /tmp/codex-lab08

ls /tmp/codex-lab08 2>/dev/null \
&& echo 'ERROR: aún existe' \
|| echo 'OK: limpio'
```

8. Resumen y siguientes pasos

8.1 Conceptos clave

Concepto	Detalle
/review	Comando que lanza reviewer dedicado: branch, uncommitted, commit, custom
Modos de review	4 modos: against branch, uncommitted, commit SHA, custom instructions
review_model	Modelo específico para /review en config.toml
Code Review GitHub	@codex review en PRs + AGENTS.md Review guidelines
Prioridades	P0 (bloqueante), P1 (importante). Solo P0/P1 en GitHub por defecto
Refactor incremental	Strangler pattern + green baseline + safe steps + invariante
Optimización con baseline	Medir → optimizar → medir → aceptar solo si mejora
Perfiles	review-only (read-only) vs. dev-edit (workspace-write)
GitHub Action	openai/codex-action@v1 para reviews automatizadas en CI
Flujo disciplinado	/review → fix → /review (confirmar resolución)

8.2 Preparación para el Módulo 9

En el Módulo 9 abordaremos Codex Cloud y Non-interactive Mode: cómo delegar tareas a entornos remotos, ejecutar Codex en CI/CD pipelines y escalar el uso del agente. Para prepararse:

- Revisar: <https://developers.openai.com/codex/cloud/>
- Revisar: <https://developers.openai.com/codex/noninteractive/>
- Revisar: <https://developers.openai.com/codex/github-action/>

9. Referencias y recursos

- Slash commands (/review): <https://developers.openai.com/codex/cli/slash-commands/>
- CLI features: <https://developers.openai.com/codex/cli/features/>
- Workflows (review): <https://developers.openai.com/codex/workflows/>
- Code Review en GitHub: <https://developers.openai.com/codex/cloud/code-review/>
- Build Code Review with SDK:
https://developers.openai.com/cookbook/examples/codex/build_code_review_with_codex_sdk/
- GitHub Action: <https://developers.openai.com/codex/github-action/>
- Config reference (review_model): <https://developers.openai.com/codex/config-reference/>
- App review pane: <https://developers.openai.com/codex/app/review/>