PLAN DE FORMACIÓN

OpenAI Codex para Desarrolladores

Aprovechamiento integral del asistente de IA para desarrollo de software

Versión 1.0 — Febrero 2026

Modelos de referencia: GPT-5.3-Codex / GPT-5.2-Codex

Superficies: Codex App (macOS) · Codex CLI · IDE Extension (VS Code / JetBrains) · Codex Cloud

Índice de contenidos

Resumen ejecutivo

Este plan de formación está diseñado para que equipos de desarrollo de software adquieran las competencias necesarias para integrar OpenAI Codex en sus flujos de trabajo diarios. Codex es el agente de programación con IA de OpenAI, capaz de escribir código, comprender bases de código desconocidas, revisar código, depurar problemas y automatizar tareas de desarrollo. El plan cubre desde los fundamentos del ecosistema Codex hasta técnicas avanzadas de integración en pipelines CI/CD y migraciones de código legacy.

La formación se estructura en 10 módulos progresivos, cada uno con objetivos claros, contenidos teóricos, buenas prácticas, errores comunes, casos de uso reales y al menos un laboratorio práctico con instrucciones paso a paso. Está orientado a un entorno profesional o corporativo y puede impartirse en formato presencial, remoto o autoguiado.

Audiencia y prerrequisitos

- Desarrolladores de software (junior a senior) con experiencia en al menos un lenguaje de programación.
- Conocimiento básico de Git y línea de comandos.
- Cuenta de ChatGPT Plus, Pro, Business, Edu o Enterprise activa.
- Node.js v22+ instalado (para Codex CLI).
- VS Code o JetBrains IDE instalado.

Duración estimada

Módulo	Tema	Duración
M1	Ecosistema Codex y modelos	3 h
M2	Instalación y configuración del entorno	3 h
M3	Prompt engineering para programación	4 h
M4	AGENTS.md y gestión del contexto	3 h
M5	Skills: diseño e implementación	4 h
M6	Generación y refactorización de código	4 h
M7	Testing y debugging asistido por IA	4 h
M8	Documentación automática y revisión de código	3 h
M9	Seguridad, privacidad y control de calidad	3 h
M10	Integración CI/CD y automatizaciones avanzadas	4 h

Total estimado: 35 horas (adaptable según nivel del grupo).

Módulo 1: Ecosistema Codex y selección de modelos

Objetivos de aprendizaje

- Comprender la arquitectura del ecosistema Codex: App, CLI, IDE Extension y Cloud.
- Conocer la familia de modelos Codex y sus diferencias de capacidad, velocidad y coste.
- Seleccionar el modelo adecuado según el caso de uso (sesión interactiva vs. tarea autónoma de larga duración).
- Entender los planes de suscripción y límites de uso.

Contenidos teóricos

1.1 Qué es Codex

Codex es el agente de programación con IA de OpenAI. A diferencia de un simple autocompletado, Codex opera como un agente completo: puede leer ficheros, ejecutar comandos, crear y modificar código, ejecutar tests y proponer pull requests. Funciona en un sandbox seguro y puede trabajar en múltiples tareas en paralelo.

1.2 Superficies de Codex

Superficie	Descripción	Plataforma
Codex App	Aplicación de escritorio con worktrees, threads paralelos y panel de revisión	macOS (Apple Silicon)
Codex CLI	Agente en terminal. Sandbox OS-level (Seatbelt/Landlock). Open source.	macOS, Windows, Linux
IDE Extension	Panel integrado en VS Code, Cursor, Windsurf y JetBrains (Rider, IntelliJ, PyCharm, WebStorm)	macOS, Linux (Windows experimental)
Codex Cloud	Entorno en la nube accesible desde chatgpt.com/codex. Tareas en background.	Navegador / iOS

1.3 Familia de modelos Codex (febrero 2026)

Modelo	Características	Uso recomendado
GPT-5.3-Codex	Más capaz. Combina rendimiento de codificación frontera con razonamiento avanzado. 25% más rápido que GPT-5.2-Codex.	Modelo por defecto para la mayoría de tareas
GPT-5.2-Codex	Modelo avanzado para ingeniería real. Context compaction y ciberseguridad.	Tareas complejas, refactors masivos
GPT-5.1-Codex-Max	Optimizado para tareas de larga duración (7+ horas autónomas).	Migraciones grandes, tareas de fondo
GPT-5.1-Codex-Mini	Versión más pequeña y económica.	Tareas rápidas, presupuesto limitado
GPT-5 / GPT-5.1	Modelos generales agentísticos con buenas capacidades de código.	Tareas mixtas código + conocimiento general

Nota: Codex permite apuntar a cualquier modelo compatible con la API Chat Completions o Responses. Se puede configurar proveedores como Azure OpenAI, Ollama (modelos OSS locales) u otros.

1.4 Niveles de razonamiento (reasoning effort)

Codex permite ajustar el esfuerzo de razonamiento entre low, medium, high y xhigh. Medium es el valor por defecto y ofrece el mejor balance entre inteligencia y velocidad. Para tareas complejas (refactors masivos, debugging profundo), se recomienda high o xhigh. Para consultas simples, low acelera la respuesta.

Buenas prácticas

- Empezar siempre con GPT-5.3-Codex como modelo por defecto; escalar a modelos Max solo para tareas de larga duración.
- Usar reasoning effort medium para interacción diaria; subir a high solo cuando la tarea lo justifique (consume más tokens y rate limits).
- En entornos corporativos, estandarizar el modelo y reasoning effort vía config.toml compartido.

Errores comunes

- Usar siempre xhigh “por si acaso”: agota rate limits rápidamente sin mejora perceptible en tareas sencillas.
- Ignorar los modelos Mini para tareas triviales, generando costes innecesarios con API key.
- No verificar que el modelo seleccionado está disponible en la superficie utilizada.

Casos de uso reales

- Equipo de backend que usa GPT-5.3-Codex para sesiones interactivas de pair programming y delega migraciones de esquemas de BD a GPT-5.1-Codex-Max en modo cloud autónomo.
- Startup que usa GPT-5.1-Codex-Mini vía API para generar scaffolding de microservicios, controlando costes.

Laboratorio 1: Exploración del ecosistema Codex

Objetivo: Instalar Codex CLI, autenticarse, verificar modelos disponibles y ejecutar una primera tarea.

Pasos

1. Instalar Codex CLI: npm i -g @openai/codex
2. Lanzar codex en terminal. Autenticarse con cuenta ChatGPT o API key.
3. Ejecutar /model para ver el modelo actual. Cambiar a gpt-5.3-codex con: codex -m gpt-5.3-codex
4. Crear un directorio temporal: mkdir /tmp/lab01 && cd /tmp/lab01 && git init
5. Pedir a Codex: “Crea un fichero hello.py que imprima ‘Hola Codex’ y un test unitario con pytest”.
6. Verificar que Codex genera los ficheros, ejecuta el test y reporta el resultado.
7. Probar con reasoning effort low: codex -m gpt-5.3-codex -c reasoning.effort='"low"' y repetir la misma tarea. Comparar tiempos.

Resultado esperado: Dos ficheros (hello.py, test_hello.py) generados, test pasando. Diferencia de tiempo medible entre reasoning efforts.

Limpieza: `rm -rf /tmp/lab01`

Módulo 2: Instalación, configuración y herramientas del entorno

Objetivos de aprendizaje

- Configurar Codex CLI, la extensión IDE y Codex Cloud de forma óptima.
- Dominar el fichero config.toml: capas de configuración, perfiles y precedencia.
- Configurar modos de sandbox y políticas de aprobación.
- Integrar Codex con VS Code y JetBrains IDEs.

Contenidos teóricos

2.1 Arquitectura de configuración

Codex resuelve la configuración en capas con el siguiente orden de precedencia (mayor a menor): flags de CLI (-c, --model) > config.toml de proyecto (.codex/config.toml, del root al CWD) > config.toml de usuario (~/.codex/config.toml) > defaults del sistema. Las capas de proyecto solo se cargan si el proyecto está marcado como trusted.

2.2 Fichero config.toml esencial

Estructura mínima recomendada para ~/.codex/config.toml:

```
model = "gpt-5.3-codex" approval_policy = "on-request" sandbox_mode = "workspace-write"  [sandbox_workspace_write] network_access = false  [features] shell_tool = true
```

2.3 Modos de sandbox

Modo	Permisos	Recomendación
read-only	Solo lectura del sistema de ficheros	Explorar codebases, Q&A sobre código
workspace-write	Escritura limitada al directorio de trabajo. Red deshabilitada por defecto.	Uso diario de desarrollo (recomendado)
danger-full-access	Sin sandbox. Acceso total a filesystem y red.	Solo en contenedores aislados / CI

2.4 Políticas de aprobación

Política	Comportamiento
untrusted	Solo comandos read-only seguros se auto-ejecutan; todo lo demás requiere aprobación
on-request	El modelo decide cuándo pedir permiso (por defecto)
on-failure	Auto-ejecuta en sandbox; pide aprobación solo si falla
never	Nunca pide (PELIGROSO — solo en entornos controlados)

2.5 Extensión IDE para VS Code

La extensión oficial (marketplace: openai.chatgpt) se instala desde el Marketplace de VS Code. Comparte la misma configuración config.toml que el CLI. Ofrece tres modos: Chat (solo

conversación), Agent (lee ficheros, ejecuta comandos, escribe cambios con aprobación) y Agent Full Access (sin aprobaciones). Permite referenciar ficheros con @fichero.ts, arrastrar imágenes al prompt y delegar tareas al cloud desde el IDE.

2.6 JetBrains y otros IDEs

La integración con JetBrains (Rider, IntelliJ, PyCharm, WebStorm) soporta autenticación con ChatGPT, API key o suscripción JetBrains AI. Los editores compatibles con VS Code como Cursor y Windsurf también son soportados.

Buenas prácticas

- Mantener un config.toml de proyecto (.codex/config.toml) versionado en el repositorio para estandarizar el comportamiento del equipo.
- Usar perfiles ([profiles.ci], [profiles.dev]) para alternar configuraciones según el contexto.
- Crear checkpoints de Git antes y después de cada tarea para poder revertir fácilmente.
- En Windows, usar WSL2 para mejor compatibilidad. Mantener repos bajo ~/code/ en Linux, no en /mnt/c/.

Errores comunes

- No marcar el proyecto como trusted: Codex ignora el config.toml del proyecto silenciosamente.
- Usar danger-full-access en la máquina de desarrollo local sin aislamiento externo.
- Olvidar que network_access = false es el default: comandos como pip install o npm install fallarán.

Casos de uso reales

- Equipo enterprise con config.toml versionado que establece approval_policy = on-request y sandbox_mode = workspace-write como estándar corporativo, con requirements.toml gestionado via MDM.
- Desarrollador individual que usa perfiles: --profile ci con sandbox completo para CI, y el perfil por defecto interactivo para desarrollo.

Laboratorio 2: Configuración del entorno de desarrollo

Objetivo: Configurar Codex CLI y la extensión VS Code con un proyecto de ejemplo, perfiles y políticas de sandbox.

Pasos

1. Crear directorio: mkdir /tmp/lab02 && cd /tmp/lab02 && git init
2. Crear ~/.codex/config.toml con modelo gpt-5.3-codex, approval_policy on-request, sandbox workspace-write.
3. Crear .codex/config.toml del proyecto con model_provider personalizado si se usa Azure.
4. Verificar configuración efectiva: codex config show --effective
5. Instalar extensión Codex en VS Code. Abrir el directorio. Verificar que el panel Codex aparece.
6. Cambiar entre modos Chat, Agent y Agent (Full Access). Observar las diferencias.
7. Probar /status en CLI para ver directorios del workspace, modelo activo y sandbox.

Resultado esperado: config show muestra la cadena de configuración resuelta. La extensión VS Code está autenticada y muestra el panel funcional.

Limpieza: `rm -rf /tmp/lab02; rm ~/.codex/config.toml  # restaurar config original si se tenía`

Módulo 3: Prompt engineering aplicado a programación

Objetivos de aprendizaje

- Dominar técnicas de prompt engineering específicas para generación de código.
- Entender cómo el contexto (ficheros abiertos, selecciones, @file) afecta la calidad de las respuestas.
- Aplicar la guía oficial de prompting de Codex para obtener resultados óptimos.

Contenidos teóricos

3.1 Principios fundamentales del prompting para Codex

Codex funciona como un agente autónomo, no como un chatbot. Los prompts más efectivos son directivos y orientados a acción, no conversacionales. Según la guía oficial de OpenAI:

- Sesgo hacia la acción: pedir implementaciones concretas, no explicaciones teóricas.
- Especificar inputs y outputs explícitos: “Recibe un JSON con campos X, Y y devuelve un array filtrado por Z.”
- Evitar preámbulos y planes extensos: el modelo rinde mejor cuando va directo a la implementación.
- Usar contexto de ficheros: en el IDE, @archivo.ts inyecta su contenido; en CLI, Codex lee automáticamente el directorio de trabajo.

3.2 Patrones de prompts efectivos

Patrón	Ejemplo	Cuándo usarlo
Tarea directa	Implementa un endpoint POST /users que valide email, hashee password con bcrypt y persista en PostgreSQL	Funcionalidades nuevas bien definidas
Refactorización con restricciones	Refactoriza este módulo para usar inyección de dependencias sin cambiar la API pública. Mantén todos los tests verdes.	Mejoras de arquitectura
Bug fix contextual	El test test_payment_flow falla con TimeoutError en la línea 47. Diagnostica la causa raíz y propón un fix.	Debugging guiado
Exploración de codebase	Explica cómo fluye una request desde routes/api.ts hasta el servicio de notificaciones. Lista los middlewares involucrados.	Onboarding a nuevo proyecto
Generación con ejemplo	Genera tests de integración para UserService siguiendo el patrón de @tests/OrderService.test.ts	Consistencia con patrones existentes

3.3 Gestión del contexto

El contexto en Codex tiene múltiples fuentes: el prompt del usuario, los ficheros AGENTS.md (ver Módulo 4), las skills activas (ver Módulo 5), los ficheros abiertos en el IDE, y el historial de la conversación. Codex soporta hasta 192k tokens de contexto (según modelo) y dispone de context compaction para sesiones largas, que resume automáticamente el historial cuando se acerca al límite.

Estrategias para gestionar el contexto eficientemente:

- Usar @file para inyectar solo los ficheros relevantes, no todo el proyecto.
- En sesiones largas, el mid-turn steering permite enviar instrucciones adicionales mientras Codex trabaja (pulsando Enter durante la ejecución).
- Usar Tab para encolar prompts de follow-up sin interrumpir la ejecución actual.
- El comando /fork permite bifurcar una sesión para explorar caminos alternativos.

3.4 Web search integrado

Codex incluye búsqueda web integrada. Por defecto usa un índice cacheado por OpenAI (menor riesgo de prompt injection). Con --search o web_search = "live" se obtienen resultados en tiempo real. Útil para consultar documentación actualizada de librerías.

Buenas prácticas

- Escribir prompts imperativos con inputs/outputs explícitos.
- Incluir restricciones de type safety: “Evita as any, usa tipos estrictos.”
- Aplicar DRY en los prompts: “Busca helpers existentes antes de crear nuevos.”
- Para tareas complejas, subir reasoning effort a high en lugar de escribir prompts ultra-detallados.

Errores comunes

- Prompts vagos como “Mejora este código” sin criterios específicos.
- Sobrecargar el contexto con ficheros irrelevantes usando @file de forma indiscriminada.
- No aprovechar el mid-turn steering: esperar a que Codex termine en lugar de redirigirlo cuando toma un camino incorrecto.

Laboratorio 3: Prompt engineering práctico

Objetivo: Practicar diferentes patrones de prompting y medir la calidad del código generado.

Pasos

1. Crear proyecto: mkdir /tmp/lab03 && cd /tmp/lab03 && git init && npm init -y
2. Abrir en VS Code con extensión Codex activa.
3. Ejercicio A (prompt vago): Pedir “Haz una API REST”. Evaluar la ambigüedad del resultado.
4. Ejercicio B (prompt específico): Pedir “Implementa una API REST con Express y TypeScript. Endpoint GET /tasks que devuelva un array de Task {id: number, title: string, done: boolean}. Incluye validación con Zod y tests con Vitest.” Comparar calidad.
5. Ejercicio C: Con un fichero existente abierto, usar @package.json en el prompt para que Codex respete las dependencias existentes.
6. Ejercicio D: Mientras Codex genera código, usar mid-turn steering (Enter) para añadir: “Añade también un middleware de logging con timestamp.”

Resultado esperado: Diferencia notable de calidad entre prompts vagos y específicos. El mid-turn steering modifica la salida en tiempo real.

Limpieza: `rm -rf /tmp/lab03`

Módulo 4: AGENTS.md y gestión del contexto persistente

Objetivos de aprendizaje

- Dominar el sistema de instrucciones persistentes AGENTS.md.
- Diseñar una jerarquía de instrucciones global → proyecto → directorio.
- Entender el mecanismo de descubrimiento y precedencia de instrucciones.

Contenidos teóricos

4.1 Cadena de descubrimiento de AGENTS.md

Codex construye una cadena de instrucciones al iniciar cada sesión, concatenando ficheros desde el ámbito global hasta el directorio de trabajo actual:

1. Ámbito global: ~/.codex/AGENTS.override.md (si existe) o ~/.codex/AGENTS.md.
2. Ámbito proyecto: desde el root del repo Git hasta el CWD, busca en cada directorio AGENTS.override.md > AGENTS.md > nombres alternativos configurados en project_doc_fallback_filenames.

Los ficheros más cercanos al CWD aparecen últimos en el prompt, por lo que su contenido tiene mayor peso. El límite por defecto es 32 KiB (project_doc_max_bytes).

4.2 Estructura recomendada de AGENTS.md

Ejemplo de AGENTS.md para el root de un repositorio:

```
# AGENTS.md  ## Convenciones del proyecto - Lenguaje: TypeScript strict - Tests: vitest. Ejecutar `npm test` antes de cada PR. - Linter: `npm run lint` debe pasar sin errores. - No añadir dependencias de producción sin aprobación.  ## Estructura - src/: código fuente - tests/: tests unitarios y de integración - docs/: documentación técnica
```

4.3 Overrides por directorio

Para equipos con normas específicas por servicio, se usan overrides anidados:

```
# services/payments/AGENTS.override.md ## Reglas del servicio de pagos - Usar `make test-payments` en lugar de `npm test`. - No rotar API keys sin notificar al canal de seguridad.
```

Buenas prácticas

- Versionar AGENTS.md en el repositorio para que todo el equipo comparta las mismas instrucciones.
- Mantener las instrucciones concisas y accionables: comandos exactos, no directrices vagas.
- Usar AGENTS.override.md solo en directorios con reglas que contradicen las del root.
- Incluir siempre los comandos de test y lint específicos del proyecto.

Errores comunes

- AGENTS.md vacío o con solo comentarios: Codex lo ignora silenciosamente.
- Instrucciones demasiado extensas que superan los 32 KiB: se truncan perdiendo las últimas secciones.

- Confiar en que AGENTS.md sustituye al prompt: es contexto complementario, no reemplazo del prompt explícito.

Laboratorio 4: Jerarquía de AGENTS.md

Objetivo: Crear una jerarquía de AGENTS.md y verificar que Codex respeta las instrucciones a cada nivel.

Pasos

1. mkdir -p /tmp/lab04/services/api && cd /tmp/lab04 && git init
2. Crear AGENTS.md en root: indicar que se use Python 3.12, pytest y black como formatter.
3. Crear services/api/AGENTS.override.md: indicar que en este directorio se use FastAPI y httpx para tests.
4. Desde /tmp/lab04, lanzar codex y pedir: “Crea un script de ejemplo con tests”. Verificar que usa Python + pytest.
5. cd services/api, lanzar codex y pedir la misma tarea. Verificar que usa FastAPI + httpx.
6. Ejecutar codex y preguntar: “¿Qué instrucciones tienes cargadas?” para verificar la cadena.

Resultado esperado: Codex adapta su comportamiento según el directorio desde el que se ejecuta, respetando la jerarquía de instrucciones.

Limpieza: `rm -rf /tmp/lab04`

Módulo 5: Skills — diseño e implementación

Objetivos de aprendizaje

- Entender el concepto de Agent Skills como paquetes reutilizables de capacidades.
- Crear skills personalizados con SKILL.md, scripts y recursos.
- Instalar, gestionar e invocar skills desde CLI e IDE.
- Diferenciar entre skills del sistema, curados, experimentales y personalizados.

Contenidos teóricos

5.1 Qué es un Skill

Un skill es un directorio con un fichero SKILL.md obligatorio y carpetas opcionales (scripts/, references/, assets/). Empaqueta instrucciones, scripts ejecutables y recursos que Codex carga bajo demanda. Los skills usan progressive disclosure: Codex lee solo el nombre y descripción al inicio, y carga las instrucciones completas solo cuando decide usar el skill.

5.2 Estructura de un skill

```
my-skill/   SKILL.md          # Obligatorio: instrucciones + metadatos   scripts/
# Opcional: código ejecutable   references/        # Opcional: documentación
assets/            # Opcional: plantillas, recursos
```

El SKILL.md usa frontmatter YAML:

```
--- name: deploy-checker description: Verifica pre-requisitos antes de un deploy.
Activar cuando el usuario pida hacer deploy o verificar readiness. --- ##
Instrucciones 1. Ejecutar `npm run build` y verificar que no hay errores. 2.
Ejecutar `npm test` y verificar cobertura > 80%. 3. Verificar que no hay secrets
hardcodeados con `grep -r "API_KEY" src/`.
```

5.3 Ubicaciones de skills

Ubicación	Alcance	Uso
~/.codex/skills/.system/	Sistema (built-in)	skill-creator, plan, skill-installer
~/.codex/skills/ o ~/.agents/skills/	Usuario global	Skills personales disponibles en todos los proyectos
.codex/skills/ o .agents/skills/	Proyecto (repo)	Skills del equipo, versionados en Git

5.4 Invocación

- Explícita: escribir $nombre-del-skill en el prompt o usar /skills para listarlos.
- Implícita: Codex selecciona automáticamente un skill cuando la descripción coincide con la tarea.
- Instalación remota: $skill-installer install <nombre> desde el catálogo oficial o desde URL de GitHub.

5.5 Configuración avanzada con openai.yaml

Para definir nombre visible, icono, color de marca, prompt por defecto y dependencias MCP:

```
# agents/openai.yaml interface:   display_name: "Deploy Checker"   icon_small:
"./assets/icon.svg"   brand_color: "#22C55E"   default_prompt: "Verifica que
```

```
estamos listos para deploy" dependencies:   tools:     - type: "mcp"       value:
"github"       description: "GitHub API para verificar PR status"
```

Buenas prácticas

- Un skill, una responsabilidad. Evitar skills que hagan demasiadas cosas.
- Escribir descripciones claras con scope y boundaries para la invocación implícita.
- Preferir instrucciones sobre scripts a menos que se necesite comportamiento determinista.
- Testear prompts contra la descripción del skill para verificar que se activa correctamente.

Errores comunes

- Descripciones genéricas (“Un skill útil”) que causan activación implícita errática.
- No reiniciar Codex después de añadir o modificar un skill.
- Crear skills con dependencias MCP no configuradas en el config.toml.

Laboratorio 5: Crear un skill personalizado

Objetivo: Diseñar, crear, instalar y probar un skill que automatice la creación de componentes React siguiendo un patrón específico.

Pasos

1. mkdir -p /tmp/lab05/.codex/skills/react-component && cd /tmp/lab05 && git init
2. Crear .codex/skills/react-component/SKILL.md con: name: react-component, description que indique cuándo activarse, instrucciones para generar componente + test + storybook.
3. Crear .codex/skills/react-component/references/template.tsx con una plantilla de ejemplo.
4. Lanzar codex. Verificar con /skills que el skill aparece.
5. Invocar explícitamente: $react-component “Crea un componente Button con variantes primary/secondary/danger”.
6. Invocar implícitamente: “Crea un componente Card reutilizable” y verificar que Codex selecciona el skill automáticamente.

Resultado esperado: El skill genera ficheros consistentes (componente .tsx, test .test.tsx, story .stories.tsx) siguiendo la plantilla y las instrucciones definidas.

Limpieza: `rm -rf /tmp/lab05`

Módulo 6: Generación, refactorización y migración de código

Objetivos de aprendizaje

- Generar código de calidad producción para frontend y backend con Codex.
- Realizar refactorizaciones complejas manteniendo la API pública.
- Planificar y ejecutar migraciones de código legacy con asistencia de IA.
- Aplicar refinamiento iterativo para mejorar progresivamente el código generado.

Contenidos teóricos

6.1 Generación de código

Codex se adapta a la estructura y convenciones del proyecto existente. Al tener acceso al filesystem completo, puede analizar patrones en el código existente y generar código coherente con la base existente. Se recomienda siempre incluir restricciones de tipo y patrones a seguir en el prompt.

Casos típicos de generación:

- Scaffolding de nuevos módulos, servicios y endpoints completos.
- Scripts de administración de sistemas (backup, migración, provisionamiento).
- Desarrollo web frontend (componentes React, Vue, Angular) y backend (Express, FastAPI, Spring).

6.2 Refactorización asistida

Codex es especialmente eficaz en refactorizaciones repetitivas y bien definidas: renombrado masivo, extracción de funciones, conversión de callbacks a async/await, aplicación de patrones de diseño. La clave es definir claramente las restricciones (“mantener la API pública”, “todos los tests deben seguir pasando”).

6.3 Migraciones de código legacy

Para migraciones a gran escala (ej. JavaScript a TypeScript, Python 2 a 3, monolito a microservicios), se recomienda usar GPT-5.1-Codex-Max o GPT-5.2-Codex con reasoning effort high en Codex Cloud. El modelo puede trabajar autónomamente durante horas, iterando sobre la implementación y corrigiendo errores de tests. Según OpenAI, se han observado sesiones autónomas de más de 7 horas.

6.4 Refinamiento iterativo

El flujo recomendado es: generar → revisar diff → solicitar ajustes → verificar tests → aplicar. Codex Cloud permite lanzar tareas en background y revisar los cambios propuestos antes de aplicarlos. Tanto la CLI como la IDE extension ofrecen previsualización de diffs.

Buenas prácticas

- Siempre solicitar que Codex ejecute los tests después de cada cambio.
- Para migraciones, empezar con un módulo pequeño como prueba piloto antes de escalar.
- Usar /review después de cada refactorización para detectar regresiones.
- Crear ramas Git dedicadas para el trabajo de Codex y revisar via PR.

Errores comunes

- Pedir migraciones masivas sin tests existentes: Codex no puede validar la corrección del resultado.
- No revisar los diffs generados y aplicar ciegamente: los agentes pueden introducir regresiones sutiles.
- Intentar migrar un proyecto entero en una sola sesión interactiva en lugar de usar Codex Cloud.

Laboratorio 6: Refactorización y migración

Objetivo: Refactorizar un módulo JavaScript con callbacks a async/await con TypeScript, manteniendo la funcionalidad.

Pasos

8. mkdir /tmp/lab06 && cd /tmp/lab06 && git init
9. Crear un fichero legacy.js con 3 funciones que usen callbacks anidados (callback hell) y un test básico.
10. git add -A && git commit -m "initial commit with legacy code"
11. Lanzar Codex y pedir: “Refactoriza legacy.js a TypeScript con async/await. Crea el fichero como legacy.ts. Mantén la misma funcionalidad y adapta los tests. Ejecuta los tests para verificar.”
12. Revisar el diff generado con /review.
13. Si hay fallos de test, usar mid-turn steering para guiar la corrección.

Resultado esperado: Fichero legacy.ts funcional con async/await, tests adaptados y pasando. Diff limpio y revisable.

Limpieza: `rm -rf /tmp/lab06`

Módulo 7: Testing y debugging asistido por IA

Objetivos de aprendizaje

- Generar tests unitarios, de integración, smoke tests, de seguridad y de GUI con Codex.
- Usar Codex para diagnosticar y corregir bugs de forma sistemática.
- Integrar la generación de tests en el flujo de desarrollo diario.

Contenidos teóricos

7.1 Generación de tests

Codex puede generar distintos tipos de tests partiendo del código fuente y los patrones existentes en el proyecto:

Tipo de test	Prompt de ejemplo	Consideraciones
Unitario	Genera tests unitarios para UserService cubriendo todos los métodos públicos, incluyendo edge cases y errores.	Cubrir happy path, errores y edge cases
Integración	Crea tests de integración para el flujo completo de checkout: crear carrito → añadir items → pagar → confirmar.	Requiere fixtures y mocks de servicios externos
Smoke test	Genera smoke tests que verifiquen que todos los endpoints de la API responden con status 2xx.	Validar deploys rápidamente
Seguridad	Crea tests de seguridad que intenten SQL injection, XSS y CSRF en los endpoints de autenticación.	Complementar con herramientas SAST/DAST
GUI / E2E	Genera tests E2E con Playwright para el flujo de login: navegar → rellenar credenciales → verificar dashboard.	Requiere selectores estables y entorno de test

7.2 Debugging asistido

Codex tiene capacidad nativa de debugging: puede leer stack traces, ejecutar el código problemático, añadir logs, reproducir el error y proponer un fix. El flujo recomendado es:

8. Proporcionar el error (stack trace, log, test fallido) en el prompt.
9. Pedir diagnóstico: “Diagnostica la causa raíz de este error.”
10. Solicitar el fix: “Implementa la corrección y ejecuta los tests para verificar.”
11. Revisar con /review para asegurar que el fix no introduce regresiones.

7.3 Cobertura y calidad

Se recomienda incluir en AGENTS.md la política de cobertura del proyecto (ej. >80%) para que Codex la respete automáticamente al generar tests. Codex puede ejecutar herramientas de cobertura como istanbul/c8 y analizar los resultados.

Buenas prácticas

- Pedir siempre tests junto con la implementación, no como paso separado.
- Proporcionar tests existentes como @referencia para mantener consistencia de estilo.

- Para debugging, incluir el máximo contexto: stack trace completo, versión de dependencias, entorno.
- Usar la funcionalidad de /review para identificar bugs antes de hacer merge.

Errores comunes

- Generar tests que solo cubren el happy path sin edge cases ni manejo de errores.
- Confiar ciegamente en que los tests generados cubren toda la funcionalidad: revisar siempre la cobertura.
- No proporcionar contexto suficiente al depurar: Codex necesita ver el código relevante, no solo el error.

Laboratorio 7: Generación de tests y debugging

Objetivo: Generar una suite de tests completa para un módulo existente y depurar un bug intencionado.

Pasos

7. mkdir /tmp/lab07 && cd /tmp/lab07 && git init
8. Crear un fichero calculator.py con funciones: add, subtract, multiply, divide (con un bug: divide no maneja división por cero).
9. Pedir a Codex: “Genera tests unitarios exhaustivos para calculator.py con pytest. Incluye edge cases para todas las funciones.”
10. Ejecutar los tests. El test de división por cero debe fallar.
11. Pedir a Codex: “El test test_divide_by_zero falla. Diagnostica y corrige el bug en calculator.py. Ejecuta los tests para verificar.”
12. Verificar que todos los tests pasan y que el fix es correcto.

Resultado esperado: Suite de tests completa. Bug detectado, diagnosticado y corregido automáticamente por Codex.

Limpieza: `rm -rf /tmp/lab07`

Módulo 8: Documentación automática y revisión de código

Objetivos de aprendizaje

- Generar documentación técnica automática (docstrings, README, ADRs, API docs) con Codex.
- Utilizar la funcionalidad de code review integrada de Codex para detectar bugs, lógica incorrecta y edge cases.
- Integrar la revisión de código con IA en el flujo de pull requests.

Contenidos teóricos

8.1 Documentación automática

Codex puede generar documentación en múltiples formatos. Los casos más productivos son:

- Docstrings/JSDoc: “Añade docstrings a todas las funciones públicas de src/services/ siguiendo el formato Google style.”
- README de proyecto: “Genera un README.md completo para este repositorio incluyendo: descripción, instalación, uso, arquitectura y contribución.”
- ADRs (Architecture Decision Records): “Documenta la decisión de migrar de REST a GraphQL como un ADR.”
- Documentación de API: “Genera documentación OpenAPI/Swagger para todos los endpoints.”

8.2 Revisión de código con /review

El comando /review en CLI e IDE extension lanza un revisor dedicado que lee el diff seleccionado y produce hallazgos priorizados y accionables sin modificar el working tree. Detecta: bugs potenciales, errores lógicos, edge cases no manejados, violaciones de type safety y patrones inseguros.

También se puede invocar Codex en pull requests de GitHub usando @codex en un comentario, lo que lanza una tarea cloud de revisión.

8.3 Codex Autofix en CI

Codex puede integrarse en pipelines CI para corregir automáticamente problemas detectados por linters y analizadores estáticos.

Buenas prácticas

- Ejecutar /review antes de cada PR para detectar problemas temprano.
- Incluir en AGENTS.md las convenciones de documentación del proyecto.
- Usar @codex en PRs de GitHub para revisiones automáticas en proyectos de equipo.

Errores comunes

- Generar documentación una vez y no mantenerla actualizada: añadir la generación de docs al AGENTS.md.
- Ignorar los hallazgos de /review asumiendo que son falsos positivos sin investigar.
- Generar documentación excesivamente verbosa para código autoexplicativo.

Laboratorio 8: Documentación y revisión de código

Objetivo: Generar documentación técnica para un proyecto y ejecutar una revisión de código con /review.

Pasos

3. mkdir /tmp/lab08 && cd /tmp/lab08 && git init
4. Crear 3 ficheros Python sin docstrings (user_service.py, auth_handler.py, db_connection.py) con funciones variadas.
5. git add -A && git commit -m "code without docs"
6. Pedir a Codex: “Añade docstrings Google style a todas las funciones públicas de los 3 módulos. Genera también un README.md del proyecto.”
7. Ejecutar /review sobre los cambios generados para verificar calidad.
8. Introducir intencionadamente un bug (ej. SQL injection en db_connection.py), hacer commit y ejecutar /review. Verificar que Codex lo detecta.

Resultado esperado: Documentación completa generada. /review detecta el bug de seguridad y proporciona hallazgos priorizados.

Limpieza: `rm -rf /tmp/lab08`

Módulo 9: Seguridad, privacidad y control de calidad

Objetivos de aprendizaje

- Comprender el modelo de seguridad de Codex: sandbox, aislamiento y control de red.
- Aplicar políticas de seguridad corporativas con requirements.toml.
- Mitigar riesgos de prompt injection, exfiltración de datos y código inseguro.
- Implementar controles de calidad sobre el código generado por IA.

Contenidos teóricos

9.1 Modelo de seguridad de Codex

Codex Cloud ejecuta cada tarea en un contenedor aislado gestionado por OpenAI. El acceso a internet está deshabilitado por defecto durante la ejecución de tareas (disponible solo durante la fase de setup para instalar dependencias).

Codex CLI/IDE usa mecanismos de sandbox a nivel de OS:

- macOS: Seatbelt (sandbox-exec).
- Linux: Landlock + Bubblewrap (bwrap) para aislamiento de filesystem.
- Windows: sandbox experimental basado en restricted tokens de AppContainer.

9.2 Gobernanza empresarial

Para entornos enterprise, Codex soporta:

- requirements.toml: restricciones admin que los usuarios no pueden sobreescribir (ej. prohibir sandbox_mode = "danger-full-access" o approval_policy = "never").
- Managed defaults: valores iniciales que se aplican al arrancar Codex, gestionables via MDM.
- Telemetría OpenTelemetry: exportación de eventos de runs, tool usage, aprobaciones/denegaciones para auditoría.
- shell_environment_policy: control de variables de entorno expuestas a los comandos (filtrado de secrets).

9.3 Riesgos y mitigaciones

Riesgo	Mitigación
Prompt injection via web search	Usar web_search = "cached" (default). Tratar resultados como no confiables.
Exfiltración de secrets	network_access = false. shell_environment_policy con include_only.
Código inseguro generado	/review antes de cada merge. SAST en CI. Política de type safety en AGENTS.md.
Ejecución no autorizada	approval_policy = "on-request" o "untrusted". requirements.toml para enterprise.
Datos sensibles en prompts	Auditoría OTEL. Data controls en ChatGPT para opt-out de training.

9.4 Control de calidad del código generado

Checklist para validar código generado por Codex:

- Tests pasan y cobertura es adecuada.
- Type checking pasa (tsc, mypy, etc.).
- Linter sin errores ni warnings críticos.
- /review no reporta bugs o seguridad.
- No se han introducido dependencias innecesarias.
- No hay secrets hardcodeados.
- El diff es revisable y comprensible por un humano.

Buenas prácticas

- Nunca usar --yolo (danger-full-access sin aprobaciones) fuera de contenedores CI aislados.
- Habilitar requirements.toml en todos los equipos corporativos desde el día 1.
- Configurar log_user_prompt = false en OTEL para no registrar prompts con datos sensibles.
- Tratar el código generado por IA con el mismo nivel de revisión que el código humano.

Errores comunes

- Asumir que el sandbox protege contra todo: no previene errores lógicos ni vulnerabilidades en el código generado.
- Exponer API keys en variables de entorno sin filtrar con shell_environment_policy.
- No auditar los eventos OTEL: perder visibilidad sobre qué está haciendo Codex en el equipo.

Laboratorio 9: Auditoría de seguridad

Objetivo: Configurar políticas de seguridad restrictivas, verificar que Codex las respeta y ejecutar una auditoría básica de código generado.

Pasos

7. mkdir /tmp/lab09 && cd /tmp/lab09 && git init
8. Configurar config.toml del proyecto con: approval_policy = "untrusted", sandbox_mode = "workspace-write", network_access = false.
9. Pedir a Codex: “Instala la dependencia requests con pip.” Verificar que Codex pide aprobación y que falla por network_access = false.
10. Crear un script Python con un bug de seguridad intencionado (ej. subprocess con shell=True y input del usuario).
11. Ejecutar /review sobre el script. Verificar que Codex identifica el riesgo de seguridad.
12. Pedir a Codex que corrija el bug y re-ejecutar /review para verificar.

Resultado esperado: Las políticas de sandbox bloquean acciones no autorizadas. /review detecta y permite corregir vulnerabilidades de seguridad.

Limpieza: `rm -rf /tmp/lab09`

Módulo 10: Integración CI/CD, automatizaciones y MCP

Objetivos de aprendizaje

- Integrar Codex en pipelines CI/CD con GitHub Actions y Codex Autofix.
- Configurar automatizaciones (triaje de issues, monitorización de alertas, gestión CI/CD).
- Conectar Codex con servicios externos mediante MCP (Model Context Protocol).
- Usar Codex en modo no interactivo para scripting y automatización.

Contenidos teóricos

10.1 Codex en CI/CD

Codex se integra en pipelines CI de varias formas:

- codex exec: modo no interactivo para ejecutar tareas desde scripts. Ejemplo: codex exec "Run tests and fix failures" --json
- Codex Cloud via GitHub Action: permite lanzar tareas cloud desde workflows de GitHub Actions.
- @codex en PRs: mencionar @codex en un comentario de PR para que revise o proponga cambios.
- Codex Autofix: corrección automática de issues detectados por linters/SAST en CI.

10.2 Automatizaciones

Codex puede actuar sin instrucciones explícitas mediante automatizaciones, gestionando tareas rutinarias:

- Clasificación automática de issues (triaje por prioridad, asignación a equipos).
- Monitorización de alertas y propuesta de fixes.
- Gestión de CI/CD: re-ejecutar pipelines fallidos, analizar logs de error.

10.3 Model Context Protocol (MCP)

MCP permite conectar Codex con herramientas y servicios externos. Se configuran servidores MCP en config.toml:

```
[mcp_servers.github] type = "streamable_http" url = "https://api.github.com/mcp"
[mcp_servers.linear] type = "streamable_http" url = "https://linear.app/mcp"
```

Los skills pueden declarar dependencias MCP en agents/openai.yaml para que Codex sepa qué herramientas necesita.

10.4 Integraciones nativas

Codex tiene integraciones oficiales con:

- GitHub: PRs, issues, code review con @codex, Codex Cloud tasks.
- Slack: delegación de tareas desde canales.
- Linear: gestión de tickets y tareas de desarrollo.

10.5 Slash commands personalizados

Además de los built-in (/review, /fork, /plan, /skills, /model, /status), se pueden crear slash commands personalizados para flujos repetitivos específicos del equipo.

Buenas prácticas

- Usar codex exec con --json para parsear resultados programáticamente en scripts CI.
- Configurar entornos de Codex Cloud específicos para CI con políticas restrictivas.
- Auditar conexiones MCP y restringir servidores permitidos con mcp_server_allowlist en requirements.toml.
- Probar automatizaciones en staging antes de producción.

Errores comunes

- Ejecutar Codex en CI con --yolo sin aislamiento de contenedor externo.
- No monitorizar el consumo de tokens/rate limits cuando Codex se ejecuta en pipelines automatizados.
- Configurar MCP servers sin autenticación adecuada, exponiendo APIs internas.

Laboratorio 10: CI/CD y automatización

Objetivo: Configurar una GitHub Action que utilice Codex para revisar automáticamente cada PR y ejecutar Codex en modo no interactivo.

Pasos

7. mkdir /tmp/lab10 && cd /tmp/lab10 && git init
8. Crear un proyecto Node.js simple con un test que pase y un AGENTS.md.
9. Crear .github/workflows/codex-review.yml que ejecute codex exec "Review the latest changes and report findings" --json en cada push.
10. Simular la ejecución local: codex exec "Run tests and report results" --json y parsear el output.
11. Crear un slash command personalizado: mkdir -p .codex/commands/deploy-check/ y definir el comando.
12. Probar: /deploy-check en Codex CLI y verificar que ejecuta el flujo definido.

Resultado esperado: Codex ejecuta tareas en modo no interactivo con output JSON parseable. El slash command personalizado funciona correctamente.

Limpieza: `rm -rf /tmp/lab10`

Anexo A: Referencia rápida de comandos y atajos

Comandos CLI principales

Comando	Descripción
codex	Lanza la interfaz interactiva (TUI)
codex exec "prompt"	Ejecución no interactiva
codex --full-auto	Modo workspace-write + on-request
codex -m gpt-5.3-codex	Seleccionar modelo
codex --search	Habilitar web search live
codex resume --last	Continuar la última sesión
codex fork --last	Bifurcar la última sesión
codex cloud exec "tarea"	Lanzar tarea en Codex Cloud
codex config show --effective	Ver configuración efectiva
codex features list	Listar feature flags

Slash commands en sesión

Comando	Descripción
/review	Revisar diff actual
/fork	Bifurcar sesión
/plan	Activar modo planificación
/skills	Listar skills disponibles
/model	Ver/cambiar modelo
/status	Ver estado del workspace
/feedback	Enviar feedback a OpenAI

Atajos de teclado (CLI/IDE)

Atajo	Acción
Enter (durante ejecución)	Mid-turn steering
Tab (durante ejecución)	Encolar follow-up
Esc Esc (composer vacío)	Editar mensaje anterior
Ctrl+G	Abrir editor externo para prompt largo
@ + Tab	Búsqueda fuzzy de ficheros
$ + nombre	Invocar skill explícitamente
! + comando	Ejecutar comando shell local

Anexo B: Estructura tipo de config.toml corporativo

```
# ~/.codex/config.toml (perfil corporativo) model = "gpt-5.3-codex"
approval_policy = "on-request" sandbox_mode = "workspace-write" personality =
"friendly"  [sandbox_workspace_write] network_access = false
[shell_environment_policy] include_only = ["PATH", "HOME", "LANG", "NODE_ENV"]
[features] shell_tool = true shell_snapshot = true  [profiles.ci] model = "gpt-
5.2-codex" approval_policy = "never" sandbox_mode = "danger-full-access"
[profiles.review] model = "gpt-5.3-codex" review_model = "gpt-5.3-codex"  #
requirements.toml (admin, via MDM) # [approval_policy] # allowed = ["untrusted",
"on-request", "on-failure"] # [sandbox_mode] # allowed = ["read-only", "workspace-
write"]
```

Anexo C: Recursos y documentación oficial

- Documentación oficial Codex: https://developers.openai.com/codex
- Quickstart: https://developers.openai.com/codex/quickstart
- Guía de prompting: https://developers.openai.com/cookbook/examples/gpt-5/codex_prompting_guide/
- Catálogo de skills: https://github.com/openai/skills
- Repositorio Codex CLI (open source): https://github.com/openai/codex
- Changelog: https://developers.openai.com/codex/changelog/
- Documentación AGENTS.md: https://developers.openai.com/codex/guides/agents-md/
- Documentación de seguridad: https://developers.openai.com/codex/security/
- Configuración de referencia: https://developers.openai.com/codex/config-reference/
- Extensión VS Code: https://marketplace.visualstudio.com/items?itemName=openai.chatgpt