AI 바이브/페어 개발 — CLAUDE CODE/CODEX 과정

저장소·CI 통합
codex-action · MCP · 병렬 세션

Session 8 · 저장소·CI 통합 자동화

codex-action 워크플로 · MCP 재사용 3곳 · worktree 병렬 · 승인 게이트 · 위험 감지

강사 박수현 · 🚀 젠아이랩스(GenAI Labs)

바이브코딩 2일 집중 · 프롬프트·컨텍스트·하네스 엔지니어링

PART 1

CI 통합의 필요성 — *로컬로 안 되는 것*

> 로컬 세션이 못 하는 세 가지 · Codex 와 Claude 두 진영의 공식 액션 비교.

로컬 도구가 못 하는 3가지

로컬 Claude Code · Codex 세션은 강력하지만, 다음 세 가지는 **사람이 자리를 비운 동안에도 자동으로 실행되어야** 한다.

🌙 PR 자동 리뷰

사람이 잠들거나 회의 중이어도 PR 은 열린다.

CI 가 하는 일 — 워크플로 트리거 → 리뷰 코멘트 자동 게시.

🏷️ 릴리스 노트 생성

릴리스 태그가 붙는 순간 노트가 나와야 한다.

CI 가 하는 일 — `on: release` 트리거로 changelog 자동 생성.

🌃 야간 리팩토링

사람이 있으면 방해되고, 대량 편집은 격리가 필요하다.

CI 가 하는 일 — 예약 스케줄(`schedule: cron`)로 야간 배치.

📌 **핵심 개념.** 로컬 세션은 **대화형**, CI 세션은 **배치형**이다. 배치형은 "무엇을 · 언제 · 어떤 권한으로" 실행할지를 워크플로 파일에 사전 정의해야 한다. 이 정의를 표준화한 규격이 `openai/codex-action` 이다.

출처: GitHub Actions + openai/codex-action

MCP 통합

Codex · Claude — *두 진영의 공식 액션 비교*

항목	`openai/codex-action`	`anthropics/claude-code-action`
호출 인증	GitHub Secret 에 `OPENAI_API_KEY`	GitHub Secret 에 `ANTHROPIC_API_KEY`
실행 방식	러너에 CLI 설치 → `codex exec`	러너에 CLI 설치 → `claude -p`
샌드박스	Codex 의 `sandbox` (legacy) · `permission-profile` (권장)	Claude 의 permission-mode 승계
MCP	`~/.codex/config.toml` 또는 action input	`.mcp.json` (프로젝트) · `~/.claude.json` (사용자) · action input
주요 용례	PR 리뷰 · 릴리스 노트 · 배치 리팩토링	동일

📌 **선택 기준 3.** 팀이 이미 어느 API 키를 확보했나 · 로컬에서 어느 도구를 쓰고 있나 · 사내 보안팀이 어느 벤더에 접근 승인을 냈나. **셋 중 두 개 이상 겹치는 쪽으로** 간다.

📌 이 세션은 **Codex를 중심**으로 다룬다. 워크플로 구조는 거의 동일하므로 Claude만 사용하는 팀도 동일한 구조를 그대로 적용할 수 있다.

PART 2

GitHub Actions + codex-action — *4개 워크플로*

> 액션 구조 · PR 리뷰 · 릴리스 노트 · 자동 리팩토링 · 승인 게이트·Secrets.

출처: GitHub Actions + openai/codex-action

MCP 통합

`openai/codex-action` 구조

공식 액션 이름은 `openai/codex-action`, 최신 안정판은 `v1` 태그. 러너에서 하는 일 세 가지.

1️⃣ CLI 설치

러너 이미지에 없는 상태로 시작 → Codex CLI 를 내려받는다.

2️⃣ Responses API 프록시

로컬 CLI 와 OpenAI API 사이에 인증 프록시를 띄운다.

3️⃣ `codex exec` 실행

지정된 프롬프트를 배치 모드로 돌린다.

액션의 주요 input.

input	하는 일	기본값
`openai-api-key`	OpenAI API 키 (Secret 참조)	필수
`prompt`	Codex 에게 시킬 지시	필수
`model`	모델 라우팅	`gpt-5.3-codex-spark`
`sandbox` (legacy) · `permission-profile` (권장)	`read-only` · `workspace-write` · `danger-full-access`	빈 값 (Codex CLI 기본 `workspace-write` 승계)
`codex-args`	`codex exec` 에 추가로 넘길 인자	없음

📌 **핵심 인지.** 액션은 **CLI 를 러너 안에서 그대로 돌린다.** 로컬에서 익힌 `codex exec` · sandbox · MCP 지식이 그대로 통한다. 별도 SaaS 를 배우는 것이 아니라 **같은 CLI 를 러너 컨테이너 위로 옮기는 것뿐**이다.

출처: GitHub Actions + openai/codex-action

MCP 통합

PR 자동 리뷰 — `codex-review.yml` *(트리거·권한·체크아웃)*

```
# .github/workflows/codex-review.yml
name: Codex PR review

on:
  pull_request:
    types: [opened, synchronize, reopened]

permissions:
  contents: read
  pull-requests: write

jobs:
  review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with:
          fetch-depth: 0
```

출처: GitHub Actions + openai/codex-action

MCP 통합

PR 자동 리뷰 — `codex-review.yml` *(Codex 실행 스텝)*

```
# (계속) .github/workflows/codex-review.yml
      - name: Run Codex review
        uses: openai/codex-action@v1
        with:
          openai-api-key: ${{ secrets.OPENAI_API_KEY }}
          model: gpt-5.3-codex-spark
          sandbox: read-only
          prompt: |
            이 PR 의 diff 를 검토하고 다음 세 관점의 리뷰를 남겨라.
            1) 기능 정합성 · 실패 가능한 엣지
            2) 보안 (secret · 인증 · 입력 검증)
            3) 성능 · 회귀 가능성
            중요도 상위 5 개만, 파일 경로와 라인 번호를 함께 표기.
```

📌 `sandbox` **(legacy)** · `permission-profile` **(권장).** 위 예시의 `sandbox: read-only` 는 하위 호환용 legacy 입력. 신규 워크플로에서는 아래처럼 `permission-profile` 로 대체한다 — Codex CLI 가 프로파일 이름 하나로 파일·네트워크·명령 권한 세트를 매핑한다.

```
        with:
          openai-api-key: ${{ secrets.OPENAI_API_KEY }}
          model: gpt-5.3-codex-spark
          permission-profile: read-only   # legacy sandbox: read-only 대체
```

출처: GitHub Actions + openai/codex-action

MCP 통합

PR 리뷰 — *결정 지점 3*

🛡️ `sandbox: read-only`

리뷰는 **코드를 읽기만 할 뿐 쓰지 않는다.**

실수 여지를 원천 차단하는 첫 번째 게이트.

🌐 `fetch-depth: 0`

얕은 체크아웃이면 diff 계산에 필요한 base ref 가 없다.

기준 커밋이 없어 diff 가 잘못 계산된다. 반드시 전체 이력.

🔑 `pull-requests: write`

리뷰 코멘트를 남기려면 이 권한이 필요.

`contents: read` 는 코드 열람용, 두 개는 역할이 다르다.

📌 **감각 잡기.** 세 개 중 하나만 빠져도 워크플로가 조용히 실패한다. 실패 로그가 명료하지 않아 트러블슈팅에 시간이 걸린다. **처음 붙일 때 세 개를 눈으로 대조** 한다.

출처: GitHub Actions + openai/codex-action

MCP 통합

릴리스 노트 자동 생성 — `codex-release-notes.yml`

```yaml
# .github/workflows/codex-release-notes.yml
name: Codex release notes
on:
  release:
    types: [created]
permissions:
  contents: write
jobs:
  notes:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4
        with: { fetch-depth: 0 }
      - name: Draft release notes
        uses: openai/codex-action@v1
        with:
          openai-api-key: ${{ secrets.OPENAI_API_KEY }}
          model: gpt-5.3-codex-spark
          sandbox: workspace-write
          prompt: |
            이전 태그~이번 태그 커밋 히스토리로 사용자 관점 changelog 작성.
            섹션 [기능 추가][수정][파괴적 변경][내부 개선], 각 한 줄·PR 번호 표기.
            결과를 RELEASE_NOTES.md 에 저장.
```

📌 **핵심 감각.** `on: release` 트리거는 사람이 릴리스 태그를 붙이는 그 순간 발화. Codex 가 changelog 를 짜서 `RELEASE_NOTES.md` 로 남기고, 후속 스텝에서 `gh release edit --notes-file` 로 릴리스 본문에 덧붙이는 흐름.

출처: GitHub Actions + openai/codex-action

MCP 통합

자동 리팩토링 — `codex-refactor.yml` (트리거·게이트·체크아웃)

```
# .github/workflows/codex-refactor.yml
name: Codex refactor on label

on:
  pull_request:
    types: [labeled]
    paths:
      - 'src/**'
      - 'lib/**'

jobs:
  refactor:
    if: github.event.label.name == 'codex/refactor'
    runs-on: ubuntu-latest
    permissions:
      contents: write
      pull-requests: write
    steps:
      - uses: actions/checkout@v4
```

출처: GitHub Actions + openai/codex-action

MCP 통합

자동 리팩토링 — `codex-refactor.yml` (Codex 실행 스텝)

```
# (계속) .github/workflows/codex-refactor.yml
      - name: Codex refactor
        uses: openai/codex-action@v1
        with:
          openai-api-key: ${{ secrets.OPENAI_API_KEY }}
          model: gpt-5.3-codex-spark
          sandbox: workspace-write
          prompt: |
            변경된 파일들을 대상으로 다음 규약대로 리팩토링하라.
            - 함수 길이 40 줄 초과 → 분리
            - 매직 넘버 → 상수화
            - 사이드이펙트 함수 → 순수 함수로 이관
            결과를 커밋으로 남기고 커밋 메시지에 근거를 표기.
```

출처: GitHub Actions + openai/codex-action

MCP 통합

트리거 조건 3종 — *러너·API 비용 절감*

🏷️ `types: [labeled]`

라벨이 붙는 순간에만 실행.

사람이 **명시적으로** 트리거하는 규약.

📂 `paths` 필터

`'src/**'` · `'lib/**'` 지정 경로가 바뀐 PR 만 대상.

문서 · 설정만 바뀐 PR 은 스킵.

🎯 `if` 조건

`github.event.label.name == 'codex/refactor'`

아무 라벨이 아니라 **특정 라벨**일 때만.

⚠️ **무차별 발화의 사고 사례.** 어떤 오픈소스가 codex-action 을 무조건 `on: push` 로 걸어놓았다가 리팩토링 커밋이 다시 push 를 트리거하면서 **무한 루프.** 시간당 러너 15개가 동시에 돌아 계정이 잠깐 정지된 사례.

세 조건을 한꺼번에 걸면 러너·API 비용이 눈에 띄게 준다. 무차별이 아니라 사람이 "지금 이 PR 에 codex 붙여줘" 라고 라벨을 붙일 때만 반응하는 게 팀 규약으로 안정적이다.

출처: GitHub Actions + openai/codex-action

MCP 통합

승인 게이트 — *3층 방어*

`workspace-write` 이상 모드에서 Codex 가 커밋 · 푸시 · PR 을 만들 수 있는 순간, 사람 승인 없이 리포가 바뀌는 위험이 생긴다. GitHub 이 제공하는 세 가지 게이트를 조합한다.

층	도구	방어 대상
1층 — 트리거 좁히기	`types` · `paths` · `if`	무차별 발화 방지
2층 — 환경 승인	GitHub Environment · Required reviewers	사람이 워크플로 실행 전 승인
3층 — 브랜치 보호	Branch protection rule · Required status checks	승인 없는 merge 차단

📌 **감각 잡기.** 1층은 워크플로 파일 안에서 못 박고, 2층은 Repository Settings 의 Environment 에서, 3층은 Branch protection 에서 걸린다. **세 층의 설정 위치가 서로 다르다** 는 게 헷갈리기 쉬운 지점.

출처: GitHub Actions + openai/codex-action

MCP 통합

Environment 승인 게이트 — *실전 YAML*

```yaml
jobs:
  refactor:
    runs-on: ubuntu-latest
    environment:
      name: codex-write
      url: https://github.com/${{ github.repository }}/pulls
    steps:
      # ... codex-action 단계
```

🔐 무엇이 일어나나

- 워크플로가 `codex-write` Environment 를 참조
- Repository Settings 에서 그 Environment 에 **Required reviewers** 를 걸어둠 — UI 경로 **Settings → Environments → New environment → Required reviewers 체크박스**
- 워크플로 실행 요청이 오는 순간 **지정된 리뷰어에게 승인 요청 알림**

📋 설계 원칙

- 프로덕션 배포용 Environment 와 CI 리뷰용 Environment 를 분리
- 리뷰용은 낮은 사용 한도 · 짧은 승인 대기
- 프로덕션용은 다중 리뷰어 · 승인 로그 필수

출처: GitHub Actions + openai/codex-action

MCP 통합

Secrets 관리 3원칙

🗄️ 저장

GitHub Repository Secrets 또는 **Organization Secrets.**

커밋에는 절대 넣지 않는다. 실수로 넣으면 즉시 회전(rotate).

🔗 참조

워크플로에서 `${{ secrets.OPENAI_API_KEY }}` 로만 접근.

로그에는 **자동 마스킹**된다.

🧱 격리

프로덕션 배포용 키 · CI 리뷰용 키를 분리.

리뷰용 키는 **사용 한도 낮게** 설정.

PART 3

MCP 통합 — *세 컨텍스트에서 재사용*

> MCP 재사용 3곳 · Codex 에 서버 등록 · CI 러너 안에서 · Codex Web 제약.

출처: GitHub Actions + openai/codex-action

MCP 통합

MCP 는 세 곳에서 재사용된다

한 번 만든 서버가 **세 실행 컨텍스트에서 그대로 재사용** 된다. MCP 재사용의 핵심 이점.

실행 컨텍스트	설정 위치	프로토콜
Claude Code (로컬)	`.mcp.json` (프로젝트 루트) 또는 `~/.claude.json` (사용자 홈)	stdio · streamable-http
Codex CLI (로컬 · CI)	`~/.codex/config.toml` 의 `[mcp_servers.<name>]`	stdio · streamable-http
Codex Web (chatgpt.com/codex)	웹 UI 의 MCP 서버 등록 화면	streamable-http 만

📌 **핵심 원칙.** MCP 서버는 자기 안에 인증 · 데이터 액세스 로직을 담고, **호출자는 프로토콜만 맞추면 된다.** 그래서 서버를 한 번 잘 짜 두면 위 세 곳에서 재사용된다.

📌 로컬에서 검증한 Slack 요약봇 같은 MCP 서버를 **Codex 에서도 그대로 붙인다.**

Codex 에 MCP 등록 — *stdio 서버*

Codex 의 MCP 설정은 `~/.codex/config.toml` 에 담긴다. 프로젝트 단위로 격리하고 싶으면 `.codex/config.toml` 을 리포 루트에 두되, 반드시 **신뢰된(trusted) 프로젝트** 로 등록.

```
# ~/.codex/config.toml
[mcp_servers.slack_summary]
command = "python"
args = ["-m", "slack_summary_bot.server"]
env_vars = [
  { name = "SLACK_BOT_TOKEN", source = "env" },
  { name = "SLACK_APP_TOKEN", source = "env" },
]

[mcp_servers.rag_indexer]
command = "python"
args = ["-m", "rag_indexer.server"]
env_vars = [
  { name = "OPENAI_API_KEY", source = "env" },
]
```

등록 후 확인.

```
codex mcp list                 # 등록된 서버 목록
codex mcp get slack_summary    # 특정 서버 설정 확인
```

출처: GitHub Actions + openai/codex-action

MCP 통합

Codex 에 MCP 등록 — *streamable-http 서버*

원격 SaaS 형태 MCP 서버는 URL 로 등록한다.

```
[mcp_servers.figma]
url = "https://mcp.figma.com/v1"
bearer_token_env_var = "FIGMA_MCP_TOKEN"
```

> `bearer_token_env_var` 는 **토큰 값이 아니라 env var 이름 문자열**을 담는다. 실제 값은 실행 시점에 환경 변수에서 읽어 오므로 리포·설정 파일에 secret 이 남지 않는다.

🌐 stdio vs streamable-http

- **stdio** — 로컬 서브프로세스 · 서버 코드를 직접 갖고 있을 때
- **streamable-http** — 원격 URL · SaaS · 팀 공유
- **같은 서버 로직**을 두 트랜스포트로 감싸두면 로컬·원격 모두 가능

🔧 실전 감각

- 파이썬 `mcp` SDK 는 두 트랜스포트를 **같은 서버 코드 위에서** 갈아 끼울 수 있다
- 개발은 stdio 로 · 배포는 streamable-http 로
- 인증은 서버가 맡고 클라이언트는 토큰만 실어 보낸다

📌 **핵심 감각.** 로컬에서 이미 Claude Code 에 붙여 검증한 MCP 서버를 그대로 이 파일에 적어 넣기만 하면 Codex 에서도 동작. **서버 코드는 손대지 않는다.** 클라이언트 프로토콜이 표준화돼 있어서다.

CI 에서 MCP 사용 — *워크플로 안 등록*

GitHub Actions 러너 안에서도 MCP 서버를 붙일 수 있다. 러너 홈에 `config.toml` 을 만들거나 action input 으로 넘긴다.

```
- name: Codex with MCP
  uses: openai/codex-action@v1
  env:
    SLACK_BOT_TOKEN: ${{ secrets.SLACK_BOT_TOKEN }}
    RAG_DB_URL: ${{ secrets.RAG_DB_URL }}
  with:
    openai-api-key: ${{ secrets.OPENAI_API_KEY }}
    model: gpt-5.3-codex-spark
    sandbox: workspace-write
    codex-args: |
      --config mcp_servers.slack_summary.command=python
      --config mcp_servers.slack_summary.args=["-m","slack_summary_bot.server"]
    prompt: |
      최근 24 시간의 #general 채널 요약을 Slack 요약봇 MCP 로 가져오고,
```

📌 **실전 팁 3.** - **MCP 서버 코드가 리포에 있다면** 러너에서 그 코드가 함께 체크아웃되므로 `python -m` 이 바로 뜬다. 별도 배포 필요 없다. - **원격 MCP 서버라면** `url = "https://..."` 로 등록하고 토큰은 `env` 로. - **디버깅** — 러너에서 `codex mcp list` 로 등록 스냅샷을 남기고, Codex 세션 로그를 `actions/upload-artifact` 로 밖으로 꺼낸다.

📌 **override 문법 감각.** `--config mcp_servers.<name>.args=[...]` 는 dotted key 로 TOML 트리를 짚어 값을 덮어쓴다. 리스트·객체 값은 JSON 문법(`["-m","..."]`)으로 실어 넘긴다 — CLI 인자 파서가 TOML 리터럴 대신 JSON 을 받도록 설계돼 있어서다. 따옴표·대괄호가 셸에서 깨지기 쉬우니 `codex-args: |` 다중 라인 블록 안에 그대로 붙여 두는 편이 안전하다.

Codex Web — *streamable-http 만*

`chatgpt.com/codex` 의 웹 UI 에서도 MCP 를 붙일 수 있다. 단, **stdio 서버는 못 붙인다.** 웹은 브라우저 안이라 서브프로세스를 띄울 수 없어서다. 원격에 떠 있는 streamable-http 서버만 가능.

웹 UI 등록 순서. Settings → MCP servers → Add server → URL · Bearer token 입력 → 검증. 등록된 서버는 Codex Web 세션에서 사용 가능한 도구로 자동 노출.

실무 배치 패턴.

```
로컬 개발              → stdio MCP (~/.codex/config.toml)
CI / GitHub Actions   → stdio MCP (러너 안에 코드 함께 체크아웃)
Codex Web (팀 공유)   → streamable-http MCP (원격 배포된 서버 URL)
```

📌 웹 컨텍스트가 요구하는 유일한 강제 조건이 **원격 URL·Bearer 토큰**. 로컬·CI 는 리포 안 코드로 끝나지만 Web 은 배포된 엔드포인트가 있어야 한다.

PART 4

병렬 세션 오케스트레이션 — *git worktree*

> 왜 병렬인가 · worktree 기본 · Codex 배치 · 비용 관찰.

출처: GitHub Actions + openai/codex-action

MCP 통합

왜 병렬이 필요한가 — *세 가지 격리*

바이브코딩이 손에 붙으면 자연히 여러 태스크를 동시에 굴리고 싶어진다. "이 세션은 인증 리팩토링, 저 세션은 문서 정리, 또 하나는 CI 최적화" 처럼.

📁 파일 격리

세션 A 의 편집이 세션 B 파일을 건드리지 않아야 한다.

🌿 브랜치 격리

각 세션이 자기 브랜치를 갖고 병합은 사람이 결정한다.

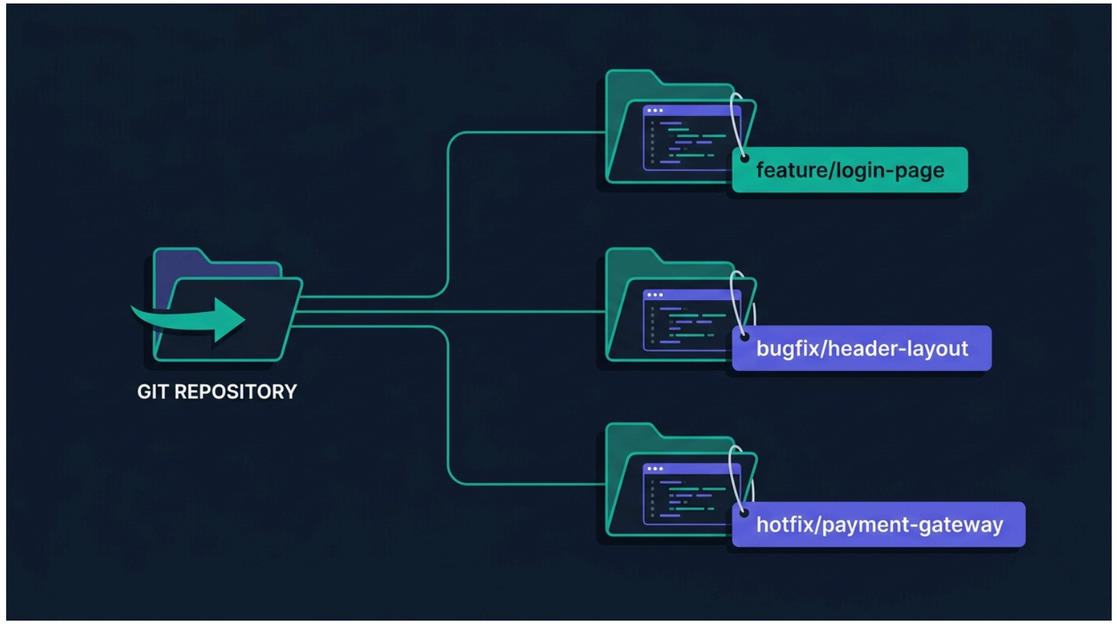
💬 컨텍스트 격리

각 세션의 대화 이력이 서로 섞이지 않아야 한다.

📌 **세 요구를 한꺼번에 만족하는 도구가** `git worktree` **.** 하나의 리포에 여러 작업 디렉토리를 붙이고, 각 디렉토리마다 다른 브랜치를 체크아웃한다.

📌 한 세션에서 태스크를 섞으면 **컨텍스트가 오염되고 리버트가 서로 엉킨다.** 이 사고를 피하기 위해 세션을 물리적으로 분리한다.

worktree — *세 격리를 한 도구로*



하나의 리포 · 세 개의 worktree · 각기 다른 브랜치

🧩 한 그림으로 요약

- **하나의 리포**에 여러 작업 디렉토리 — `.git` 은 공유된다
- **각 디렉토리**가 자기 브랜치·자기 세션 이력을 갖는다
- **세 격리**(파일 · 브랜치 · 컨텍스트)가 도구 하나로 정리된다

📌 이 그림이 이번 파트의 한 장짜리 개요. 뒤이어 표준 `git worktree add` 와 Claude Code 내장 명령을 다룬다.

출처: GitHub Actions + openai/codex-action

MCP 통합

git worktree — *기본 · Claude Code 내장*

git 자체가 오래전부터 제공한 기능이지만, 손 감각을 잡기 어려워 잘 안 쓰였다. **2026년 2월 Claude Code v2.1.49 이후 worktree 가 CLI 에 내장** 되면서 실전에서 편해졌다.

🛠️ 표준 git worktree

```
# 현재 리포에 새 worktree 추가
git worktree add \
  ../myapp-auth-refactor \
  feature/auth-refactor

# 목록
git worktree list

# 제거
git worktree remove ../myapp-auth-refactor
```

🤖 Claude Code 내장

```
# 세션을 자동으로 새 worktree 안에서 시작
claude --worktree auth-refactor

# .claude/worktrees/auth-refactor 아래에
# 격리된 작업 공간 생성
```

📌 **subagent 격리.** 세션이 여러 subagent 를 병렬로 굴릴 때, 커스텀 subagent frontmatter 에 `isolation: worktree` 를 넣으면 subagent 마다 자기 worktree 를 갖는다. 세션 안에서 "use worktrees for your agents" 지시로도 된다.

📌 **자동 정리.** worktree 는 방치되면 쌓인다. Claude Code 는 `~/.claude/settings.json` 의 `cleanupPeriodDays` (기본 30) 이후 오래된 worktree 를 자동 제거하되, uncommitted 변경 · untracked 파일 · unpushed 커밋이 있으면 남긴다.

출처: GitHub Actions + openai/codex-action

MCP 통합

Codex + worktree — *새벽 배치 3개*

```
# 사전에 worktree 3 개 준비
git worktree add ../myapp-a feature/lint-cleanup
git worktree add ../myapp-b feature/deps-upgrade
git worktree add ../myapp-c feature/docstring-fill

# 각 worktree 에서 병렬 실행
(cd ../myapp-a && codex exec --sandbox workspace-write \
   "린트 경고 전부 잡고 PR 초안") &

(cd ../myapp-b && codex exec --sandbox workspace-write \
   "종속성 패치 릴리스 반영, 브레이킹 체인지 검토") &

(cd ../myapp-c && codex exec --sandbox workspace-write \
   "docstring 누락된 public 함수 채우기") &

wait
```

⚠️ **주의 3가지.** - **API 한도** — 세 배치가 동시에 돌면 rate limit 이 걸릴 수 있다. `--fallback-model` 로 대체 모델을 지정하거나 `sleep` 을 사이에 끼운다. - **머지 충돌** — 세 브랜치가 같은 파일을 만지면 머지 시 충돌. 태스크를 파일 · 모듈 단위로 미리 쪼갠다. - **관찰 가능성** — 배치가 끝난 뒤 어느 세션이 뭘 바꿨는지 알아야 한다. `codex exec --output-format json > log-a.json` 처럼 로그를 분리 저장.

출처: GitHub Actions + openai/codex-action

MCP 통합

리소스·비용 관찰 — *지표 3종*

병렬 세션은 편의가 큰 만큼 비용도 빨리 붙는다. 세 가지 지표를 매일 확인.

지표	확인 방법	임계
API 토큰 소비	OpenAI Dashboard · `codex --show-cost`	하루 예산 90 % 도달 시 알림
러너 시간	GitHub Settings · Billing → Actions minutes	월 한도의 70 % 도달 시 검토
worktree 개수	`git worktree list` 행 수	10 개 초과 시 정리

📌 **비용 절감 4패턴.** - **모델 라우팅** — 리뷰·요약은 저비용 모델, 리팩토링·설계는 고비용 모델. 워크플로마다 `model` 을 다르게. - **트리거 좁히기** — Part 2 의 `types` · `paths` · `if` 조합. - **캐싱** — `actions/cache` 로 의존성 설치 시간 절약. codex-action 은 매 실행마다 CLI 를 다시 내려받으므로 캐싱하면 러너 시간 20~30 % 감축. - **재시도 규약** — 실패한 잡을 자동 3회 재시도. 일시적 rate limit 은 사람 개입 없이 해결. 단, 재시도마다 API 호출은 새로 나가니 예산 상한과 균형.

PART 5

승인 게이트·위험 감지 — *조밀한 안전망*

> 위험한 변경 3종 · 위험 감지 워크플로 · 승인 정책 · 사고 회피 3사례.

위험한 변경 3종 — *자동 감지 대상*

자동화가 편해질수록 사람이 놓치는 변경이 늘어난다. 특히 위험한 세 카테고리는 워크플로가 자동으로 감지해 사람 승인 게이트로 보낸다.

카테고리	감지 신호	왜 위험한가
스키마 마이그레이션	`migrations/**` · `schema.sql` · `alembic/**` 경로 변경	롤백 어렵고 데이터 손실 위험
보안 관련	`auth/**` · `crypto/**` · `secrets/**` · 정규식 `password` · `token` · `api_key`	사고 시 즉시 인시던트
파괴적 API 변경	public export 삭제 · 시그니처 변경 · 버전 major bump	하위 호환 깨짐 · 클라이언트 전면 수정

📌 **핵심 감각.** 감지 → 라벨 부착 → Environment 승인 게이트 → 지정 리뷰어 알림. 이 파이프라인이 안 걸리는 PR 은 자동 진행, **걸리는 PR 은 사람이 반드시 눈으로 본다.**

출처: GitHub Actions + openai/codex-action

MCP 통합

위험 감지 워크플로 ① — `scan` job

`scan` job 이 변경된 파일 경로를 정규식으로 훑어 `danger` outputs 을 세팅한다.

```
# .github/workflows/codex-danger-gate.yml
name: Codex danger gate
on:
  pull_request:
    types: [opened, synchronize]
jobs:
  scan:
    runs-on: ubuntu-latest
    outputs:
      danger: ${{ steps.detect.outputs.danger }}
    steps:
      - uses: actions/checkout@v4
        with: { fetch-depth: 0 }
      - id: detect
        run: |
          CHANGED=$(git diff --name-only origin/${{ github.base_ref }}...HEAD)
          echo "$CHANGED" | grep -qE '(migrations/|schema\.sql|alembic/|auth/|crypto/|secrets/)' \
            && echo "danger=true"  >> $GITHUB_OUTPUT \
            || echo "danger=false" >> $GITHUB_OUTPUT
```

📌 **핵심 지점 2.** - `outputs.danger` — 다음 job 이 받아서 조건 분기. - `fetch-depth: 0` — base ref 와 diff 를 정확히 계산하려면 전체 이력 필요.

📌 `$GITHUB_OUTPUT` **감각.** `key=value` 를 이 환경 파일에 append 하면 `steps.<id>.outputs.<key>` 로 노출되고, `outputs:` 매핑을 걸면 다른 job 이 `needs.<job>.outputs.<key>` 로 받는다 — 파일 한 줄이 job 간 값 전달의 표준 채널.

출처: GitHub Actions + openai/codex-action

MCP 통합

위험 감지 워크플로 ② — `gate` job

`gate` job 은 `scan` 결과가 `true` 일 때만 발화하며, Environment 로 리뷰어 승인을 강제한다.

```
gate:
  needs: scan
  if: needs.scan.outputs.danger == 'true'
  runs-on: ubuntu-latest
  environment:
    name: dangerous-changes
  steps:
    - name: Notify reviewers
      run: |
        echo "Dangerous change detected. Awaiting reviewer approval."
```

🔗 job 얽힘의 세 요소

- `needs: scan` — scan 이 끝난 뒤 시작
- `if: needs.scan.outputs.danger == 'true'` — outputs 조건 분기
- `environment: dangerous-changes` — Repository Settings 의 Environment 에서 Required reviewers 강제

🎯 결과 흐름

- 안전한 PR — scan 만 돌고 gate skip
- 위험한 PR — scan → gate 진입 → 리뷰어 알림 → 승인 대기
- **자동 진행 · 사람 진입의 경계가 이 워크플로 하나로 정리된다**

라벨 · CODEOWNERS · 정책

승인 게이트의 세 부품(**라벨 · 리뷰어 · 정책**)을 팀 규약으로 못 박는다.

🏷️ 라벨 설계 3원칙

- `codex/auto-approve` — 자동 병합 허용 (린트·문서·사소한 리팩토링)
- `codex/needs-review` — 사람 리뷰 필수. 위험 카테고리 감지 시 자동 부착
- `codex/blocked` — 임시 정지. 논의 중이거나 이슈 발견 시

👥 CODEOWNERS 매핑

```
# .github/CODEOWNERS
migrations/     @db-admins @backend-leads
auth/           @security-team
crypto/         @security-team
src/api/**      @api-owners
docs/           @tech-writers
```

📌 **정책 문서화 5항목.** `docs/codex-policy.md` 같은 문서에 다음 다섯 개를 명시.

1. Codex 가 자동 병합할 수 있는 조건 (라벨 · 파일 경로 · 커밋 규모)
2. 사람 승인이 필수인 조건 (위 위험 3종 · 예산 상한 초과)
3. Codex 결과물을 수용/거부하는 기준
4. 사고 발생 시 롤백 절차 (`git revert` · Environment 잠금 · Secret 회전)
5. 정책 변경 이력 (누가 · 언제 · 왜 바꿨나)

이 문서가 있어야 codex-action 이 **팀 도구로 자리잡는다.** 없으면 "누가 이거 켰어?" 로 매번 되돌아간다.

출처: GitHub Actions + openai/codex-action

MCP 통합

사고 회피 사례 3 — *실전 관찰*

자동화가 커질수록 사고도 커진다. 실전에서 반복 관찰되는 세 사례.

🔐 사례 A — Secret 노출

상황. codex-action 이 `.env` 를 참고해 배포 스크립트 작성 → 실제 토큰이 스크립트에 하드코딩 → PR diff 에 노출.

회피. - `.env` 를 `gitignore` - 러너 환경 변수로만 전달 - `gitleaks` 를 사전 훅으로

🔄 사례 B — 무한 루프

상황. `on: push` 트리거 → Codex 가 자기 커밋을 다시 push → 트리거 재발화 → 3시간에 러너 시간 200분 소비.

회피. - 트리거를 `pull_request` 로 좁히기 - 커밋 메시지에 `[skip ci]` 삽입 규약 — GitHub 이 커밋 메시지에서 이 매직 문자열을 인식하면 push 트리거 워크플로 전체를 스킵한다 - 최대 반복 횟수를 `if` 조건에 명시

🗑️ 사례 C — 대량 삭제

상황. "쓰지 않는 파일 정리" 지시 → Codex 가 실제로 쓰이는 유틸을 정적 분석만으로 판단해 삭제 → 다음 배포에서 프로덕션 500 에러.

회피. - 삭제는 별도 라벨·워크플로로 격리 - 삭제 diff 는 항상 사람 승인 - `sandbox: read-only` 로 스캔만·삭제는 이슈로

⚠️ **공통 원칙.** 자동화 사고는 사람의 실수보다 빠르게 번진다. **속도가 빠른 만큼 안전 게이트는 조밀해야 한다.**

세 사례의 공통 회피는 (a) **트리거 좁히기** (b) **사람 승인 게이트** (c) **실행 기록을 남기기 · observability**.

PART 6

미니 실습·정리 — *손에 붙이기*

> 실습 2개 (codex-action 워크플로 붙이기 · MCP 서버 등록) · 세션 정리 한 문장.

출처: GitHub Actions + openai/codex-action

MCP 통합

실습 1 — *codex-action 워크플로 붙이기*

```
# 1. 자기 사이드 리포에 파일 생성
mkdir -p .github/workflows
# codex-review.yml 을 리포에 붙인다 (앞 슬라이드의 YAML 그대로)

# 2. Secret 등록
#    GitHub 웹 UI 에서 Settings → Secrets → Actions
#    OPENAI_API_KEY 등록

# 3. 커밋 · 푸시
git add .github/workflows/codex-review.yml
git commit -m "ci: add codex PR review workflow"
git push

# 4. 아무 PR 하나 열어서 리뷰가 붙는지 관찰
```

👀 관찰 포인트

- Actions 탭에서 워크플로가 뜨는지
- 완료 후 PR 에 Codex 코멘트가 붙는지
- 실패하면 로그에서 원인 확인

🛠️ 트러블슈팅 상위 3

- **워크플로 미실행** → Secret 이름 오타
- **코멘트 미부착** → `permissions: pull-requests: write` 누락
- **YAML 문법 오류** → 들여쓰기 · 콜론 재확인

출처: GitHub Actions + openai/codex-action

MCP 통합

실습 2 — *MCP 서버 등록*

```
# ~/.codex/config.toml
[mcp_servers.echo]
command = "python"
args = ["-c", "import sys; sys.stdout.write('hello mcp')"]
```

```
codex mcp list
codex mcp get echo
```

👀 관찰 포인트

- 서버 목록에 `echo` 가 뜨는지
- `codex mcp get echo` 로 등록 설정이 뜨는지
- 로컬에서 이미 쓰던 MCP 서버가 있으면 그걸 등록해도 좋다

🎯 실전 확장

- 이미 운영 중인 Slack 요약봇 같은 서버가 있으면 그대로 등록
- 등록 후 `codex exec "채널 요약 가져와줘"` 로 호출
- **서버 코드는 손대지 않고 재사용** 감각을 잡는다

마무리 — *한 문장으로*

참여자 리포에 codex-action 이 붙어 PR 리뷰가 자동으로 뜨고, 로컬에서 만든 MCP 서버가 Codex 에서도 그대로 연결되고, worktree 로 병렬 세션을 굴린다 — 이 세 가지가 이번 세션의 도달점이다.

🎯 이 세션의 점검 기준 한 줄

"자동화의 속도만큼 게이트는 조밀하게. 트리거를 좁히고, 사람 승인을 남기고, 로그를 남긴다."

codex-action YAML 을 쓸 때, MCP 서버를 등록할 때, worktree 를 열 때 이 **세 가지**를 점검한다.

🎯 트리거

`types` · `paths` · `if` 로 좁힌다. 무차별 발화 = 무한 루프.

🔐 승인

Environment · Required reviewers · CODEOWNERS. 위험은 사람이 본다.

📋 로그

`--output-format json` · `upload-artifact` · `codex mcp list`. 무엇이 돌았는지 남긴다.