ChatGPT

Multi-Instance Codex CLI for Safe, Scalable Development

A multi-instance (multi-agent) setup can accelerate development by running specialized Codex processes in parallel (e.g. a *scaffold* agent, a *tester* agent, etc.) while isolating their access. This approach brings benefits like role separation and throughput, but also introduces complexity and risk. In the context of a modular-monolith (FastAPI + Next.js + PostgreSQL + Docker) for a small team, we explore **when and how** to safely run multiple Codex CLI instances. We cover use-cases, feature mapping, orchestration patterns (local and CI), security, audit/provenance, governance, migration rollback, and provide concrete templates (config, YAML, rules, prompts, scripts) and checklists. Recommendations are grounded in official Codex CLI docs and best practices (OWASP, GitHub Actions, SRE/CICD guidance).

Executive Summary (top recommendations) – Use a single Codex instance by default. Scale to multiple *scaffold*, *fixer*, *reviewer* agents only when justified (see decision matrix). Enforce **least privilege** by default (read-only sandbox), require human review of generated code (never auto-merge), and pin all Codex versions/prompts for reproducibility. In CI (GitHub Actions), run Codex steps in isolated jobs with per-job secrets. Always log every Codex run (JSONL output, prompts, versions) to an audit store. Provide `execpolicy` rules to block dangerous commands (e.g. destructive shell ops). Use skills (via `.agents/skills`) and AGENTS.md to encapsulate domain knowledge. Prioritise immediate actions: set up CODEX_HOME, create `AGENTS.md`, configure CI jobs for lint/test, draft basic `codex` rules, and test one Codex "scaffold" run in a feature branch.

1. Rationale & Use-Cases for Multi-Instance Codex

For a 1–3 developer team, a **single Codex CLI instance** (per task) is simplest and often sufficient. However, certain scenarios motivate multiple instances (see Table 1):

- **Role separation:** Run one agent focused on scaffolding code, another on writing tests, another on docs/reviews. Each agent uses a tailored instruction set, reducing prompt complexity.
- **Parallel tasks:** Perform code generation and concurrent checks (lint, security analysis) simultaneously, speeding up feedback cycles.
- **Isolation & security:** Limit each instance's privileges to a subset of project (e.g. front-end vs back-end directories).
- **Scaling automation:** In CI/CD, use multiple agents to process PRs, security, and release steps in parallel.

For a small project, multi-instance is **optional**. Use it when project size or pipeline complexity grows. Table 2 (later) offers signals: e.g. repeated long Codex runs, distinct code domains (finance vs dev modules), or backlog of manual tasks. As GitHub issue #3280 noted: *"We should be able to orchestrate coding agents... since each one has a dedicated domain"* [1] . In practice, teams have run *"multi-agent security pipelines that catch SQL injection before code review"* [2] , demonstrating real-world gains.

Table 1. *Use-cases for multiple Codex instances vs single-instance trade-offs.*

Scenario	Single Instance	Multiple Instances
Simple CRUD tasks	Easy setup, single prompt context.	Overkill; additional complexity.
Parallel operations	Must run tasks one-by-one (slower).	Can run simultaneously (faster feedback) [3]. But be wary: parallel heavy tasks can crash Codex [3].
Domain separation	One large prompt with mixed context.	Agents per domain (front-end vs back-end) – each uses relevant AGENTS.md guidance [4].
Security checking	Codex or external tools in pipeline.	Dedicated *security agent* can pre-scan code changes (e.g. SQLi detection as in Rezvani's example [2]).
Maintenance speed	Simpler, lower overhead.	High initial setup/coordination cost, but scales tasks.
Risk management	Fewer moving parts, easier auditing.	Need strict sandbox rules and logs per agent.

In summary, multi-instance is a **scaling** technique: use selectively. For a 1–3 person team, start single-instance (monolith-first), then evolve to parallel agents when productivity bottlenecks (e.g. code review backlog, long refactors) appear.

2. Codex CLI Features Mapped to Multi-Instance Patterns

We align Codex CLI capabilities with multi-agent workflows. Where official docs lack specifics, we note it as *"unspecified"* and suggest conservative defaults.

Interactive vs Non-Interactive

- **Interactive (TUI) mode** (just `codex`) is for exploratory, one-off sessions. Not suitable for automation in CI because it requires a terminal. Multi-instance use in CI requires **non-interactive mode** (`codex exec`) [5].
- `codex exec` is scriptable. It accepts a prompt (inline or via `-`), runs the plan/tools, and can output JSONL (via `--json`) [6]. This is essential for CI and parallel runs. Each `codex exec` is one session; to run multiple, simply launch multiple `codex exec` processes.

Sandboxing Modes

Codex supports three sandbox levels: `read-only`, `workspace-write`, `danger-full-access` [7]. For multi-instance:

- **Default to** `read-only`: safest (no file writes) for any agent that should not modify code, e.g. reviewers or linters. This enforces safety by default (low effort, high long-term benefit).
- `workspace-write`: allows editing files. Use only for scaffold/implement agents. Combine with *approval* settings to gate changes.
- `danger-full-access`: unrestricted. **Avoid** unless absolutely needed (e.g. massive refactor in single dev context) as it breaks containment. We treat it as prohibited for multi-agent CI.

Use **profiles** in config.toml to pre-set sandbox per role. Example `.codex/config.toml` (later) will show roles (dev, test, doc) with sandbox and approver settings.

AGENTS.md and Rules

- **AGENTS.md** provides global/project instruction chain [8]. For multi-instance, each agent shares the same repository, but we can use **per-agent overrides** via CODEX_HOME or working directory tricks:
- Set `CODEX_HOME` or `--profile` per instance to isolate global AGENTS.md if needed (e.g. one agent only sees global rules, another only sees project-specific).
- Each agent can be launched from a subdirectory with its own `AGENTS.override.md` to specialize instructions for that role (see [24†L295-L303]).
- **Rules and ExecPolicy**: Use `.codex/rules/*.yaml` files with allow/deny blocks. This is crucial for multi-agent safety. For example, a CI *fixer* agent should *not* delete or create PRs, so forbid `rm -rf` or `git push`. Use `codex execpolicy check` to validate rules [9]. (See templates below.)

JSONL, Version Pinning, Auth

- **JSONL output**: In CI, always run with `--json` to record every step (prompts, actions, message deltas) as newline-delimited JSON. This stream is key for audit logs. Enable `--output-schema` if expecting structured data from Codex (like JSON-based API spec).
- **Session Resume**: In CI runs, keep sessions short; no need to resume across jobs. In local dev, `codex exec resume --last` can continue a paused session, but for reproducibility avoid long-lived sessions in CI.
- **Auth modes**: In CI, use an API key with limited scope (e.g. ChatGPT API key or GPT-5 Codex key). **Unsure** if Codex CLI can use ephemeral tokens from OIDC; likely not. For now, store API key in GitHub Secrets (read-only for runners) and mount to `CODEX_API_KEY` env var. On local, either ChatGPT account login (with quotas) or API key. Document both options.
- **Version pinning**: Always pin Codex CLI version (e.g. via `brew install codex@x.y` or Docker image). Also pin model (e.g. `--model gpt-5.3-codex`). Save the CLI binary in Docker image or via GitHub Action checkout. Reference [20†L343-L352] for model usage rates.

3. Orchestration Patterns

Local Development

Per-Instance Settings

Each Codex process should use its own home/config to avoid state collision. Strategies: - Set `CODEX_HOME` per instance (e.g. `CODEX_HOME=$PWD/.codex-sandbox`). This isolates history, chats, and skills. - Use `--profile` for different roles. Example: in `.codex/config.toml` define profiles `scaffold`, `fix`, `review`, each with different `sandbox` and `credential`:

```
[profile.scaffold]
sandbox = "workspace-write"
ask_approval = "on-change"
[profile.fixer]
```

```
sandbox = "read-only"
ask_approval = "never"
[profile.reviewer]
sandbox = "read-only"
ask_approval = "on-request"
```

(Expand later in appendix.)

Launching Multiple Agents

For example, a developer on branch `feature/foo` might run:

```
# Scaffold agent (writes code)
CODEX_HOME=./codex_scaffold codex exec --profile scaffold "Create boilerplate
for feature 'foo'"
# Test-fixer agent (non-writing, reviews code)
CODEX_HOME=./codex_fixer codex exec --profile fixer "Review code in ./apps/
api for security issues"
```

These run in parallel shell terminals. Each has separate CODEX_HOME so they don't share context unless explicitly allowed. If both open windows, they won't conflict.

Per-Project / Per-Directory Skills

Place skills under the repo: e.g. `.agents/skills/{scaffold, fixer, reviewer}/SKILL.md`. Codex loads them by name or via implicit triggers [10]. For instance, a `scaffold` skill might contain templates or generators. Skills reduce repeated prompting overhead.

CI (GitHub Actions)

Use multiple jobs (in a workflow) to run Codex tasks in parallel:

- **Job matrix**: define roles. Example `matrix: [scaffold, fixer, reviewer]` each job sets `CODEX_HOME: ${{ github.workspace }}/codex_$role`.
- **Secrets per job**: restrict secrets to needed scopes. For instance, only the fixer job might have a DB read secret if running DB checks, while scaffold only needs GitHub token.
- **Ephemeral runners**: each job runs on a clean VM/container, isolating instances completely.
- **Gating and PR-only**: require runs on PR branches, not main. Disable write access (`allow-bots: false` or manual review required).
- **Artifacts**: upload JSONL logs (`--json > log.json`) and Codex `final_message.txt` to GitHub as artifacts for audit.

Example GitHub Actions snippet (simplified):

```
name: Codex Agents
on: [pull_request]
jobs:
  codex:
```

```
    strategy:
      matrix:
        role: [scaffold, fixer, reviewer]
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Codex CLI
        uses: openai/setup-codex@v1
        with:
          version: 5.3
      - name: Run Codex agent
        env:
          CODEX_HOME: ${{ github.workspace }}/codex_${{ matrix.role }}
          CODex_API_KEY: ${{ secrets.CODEX_${{ matrix.role }}_KEY }}
        run: |
          case "${{ matrix.role }}" in
            scaffold) codex exec --json --profile scaffold ... ;;
            fixer)    codex exec --json --profile fixer ... ;;
            reviewer) codex exec --json --profile reviewer ... ;;
          esac
      - name: Upload Codex logs
        uses: actions/upload-artifact@v3
        with:
          name: log-${{ matrix.role }}
          path: codex_${{ matrix.role }}/log.json
```

This job runs all three roles in parallel (one per runner). Adjust prompts and secret usage per role.

(Optional) GitLab CI snippet would be similar using `parallel` matrix in `.gitlab-ci.yml`.

Hybrid (Local + CI + Runners)

For a mix of local and CI usage: ensure consistent setups. Use the same `codex/config.toml` and `AGENTS.md` in repo. Encode any CI-only settings (like `--output-file`) in the workflow, not in code. Developers can replicate CI runs locally via a `make` target.

4. Security & Secrets

Finance apps demand high security. We recommend **conservative defaults**:

- **Secrets Storage:** Never hard-code API keys. In GitHub Actions, store keys as *repository secrets* (encrypted by GitHub). For local dev, consider using environment variables or a secrets manager (e.g. AWS Secrets Manager, Vault). OWASP advises centralizing and rotating secrets [11].
- **Least Privilege:** Always restrict Codex credentials to minimal scope. For API keys, grant only Codex access. For GitHub tokens, set default permissions to read-only unless needed [12]. Do not give Codex agent a user account with broad privileges.
- **Network Isolation:** In CI, run Codex on GitHub's runners; it cannot reach internal DB or cloud APIs unless secrets allow. For added safety, use OIDC to fetch temp creds if supported (currently unspecified).

- **Sandboxing & Rules:** Use Codex's sandboxing plus your own rules. Forbid dangerous operations in rules: e.g. disallow `sudo`, `kill`, `rm -rf /*`, `curl http://`, etc. Provide an example `rules.yaml` (below) that sets most commands to `prompt` or `deny`. The `execpolicy` command can verify these rules.
- **Prompt Injection Mitigation:** Ensure user inputs (like issue content or PR text) are not blindly fed into Codex prompts. Always escape or parameterize untrusted data, and avoid including raw user content in `AGENTS.md`. Codex CLI does not currently auto-sanitize prompts, so treat it like any interpreter: quotes, escapes, or use `--no-color` / `--output-schema` to validate.
- **Audit Logging:** Log every invocation (prompts and actions). GitHub Actions automatically record logs, but also **explicitly save the JSONL** (using `--json`). Structured logs should include at least: timestamp, agent role, CLI version, model, session ID, prompt hash, commit SHA. Store these as artifacts or push to a log store (or both).

Secrets Guidelines (OWASP/GitHub)

- *Store secrets in platform-provided vaults:* e.g. GitHub Secrets. Do not put in code or plaintext files [13].
- *Rotate frequently:* Use short-lived tokens if possible. OWASP recommends rotation and pipeline automation [14].
- *Audit usage:* Regularly check who has access to secrets; use GitHub audit logs.
- *Never echo secrets:* Use GitHub's mask or `::add-mask::` if printing is needed [15].

5. Auditability & Provenance

Each Codex run produces an event trail. We must capture and store this for auditing and reproducibility.

Logging Codex Invocations

- **JSONL Output:** Always run with `--json` to get newline-delimited JSON events [6]. Save this (e.g. `codex.log`). Upload it as CI artifact (for PR runs) and keep locally after dev runs.
- **Metadata:** Alongside JSON, record: CLI version (`codex version`), model name, `AGENTS.md` hash, rules file hash, Git commit SHA, runner (hostname/ID), and start timestamp. This can be one JSON record or YAML at start of log.
- **Example schema:**

```
{
  "runId": "uuid",
  "timestamp": "2026-02-10T12:34:56Z",
  "cliVersion": "5.3.0",
  "model": "gpt-5.3-codex",
  "agentsHash": "sha256:abcd...",
  "rulesHash": "sha256:1234...",
  "commitSHA": "9f8e7d...",
  "role": "scaffold",
  "inputs": {
     "prompt": "...", "filesChanged": []
  },
  "outputs": { ... },
```

```
  "metrics": { "tokensUsed": 2450 }
}
```

Store this with the JSON events. SLSA provenance suggests capturing similar data: builder ID, buildType, parameters, resolvedDependencies, and outputs [16] [17]. Here, treat Codex CLI as the builder: record `buildType: codex exec scaffold`, `builder.id: codex-cli-v5.3`, `parameters: {prompt:hashed, agentsHash, rulesHash}`, `subject: files output`.

Metrics & Observability

Record key metrics (as Prometheus counters or custom logs): - **Usage counts:** total Codex invocations, per agent role. - **Latency:** time per run. - **Tokens consumed** (Codex provides `/status` or count from logs). - **Outcome metrics:** number of PRs created vs updates, number of warnings/errors from Codex. - **Alerts:** if a Codex job fails or uses `danger-full-access` unexpectedly, trigger an alert to dev.

Send metrics to your monitoring (like Prometheus/Grafana). Example metric names: `codex_runs_total{role="scaffold"}`, `codex_failures_total`, `codex_tokens_consumed_total`.

6. Operational Patterns & Instance Roles

Define **instance roles** to compartmentalize responsibilities. Example roles:

- **Scaffold/Implementer:** Generates or updates code. Uses `workspace-write` sandbox. Prompt template: “Implement feature X” or scaffolding commands. Require approval (`on-change` or `always`).
- **Test-Fixer:** Focuses on tests and linting. Uses `read-only` or limited write (maybe `workspace-write` to add tests, but can `prompt` before changes). For example, runs after new code: “Write unit tests for changed files.”
- **Security Auditor:** Scans code for vulnerabilities. `read-only`. Only asks for fixes (produces a report or suggestions as output).
- **Doc/ADR generator:** Updates docs or ADRs. `workspace-write` with on-request approval. Generates architecture decisions in `docs/`.

Each role has resource considerations. Limit simultaneous heavy roles (like two "refactor-all" tasks in parallel) because GitHub issue #10887 shows that parallel heavy Codex tasks can crash [3]. Use job scheduling: e.g., queue heavy jobs sequentially.

Concurrency controls: In CI, you can use concurrency groups to prevent overlapping runs on the same target branch. In local, avoid literally simultaneous codex sessions in same directory; at least use separate CODEX_HOME to reduce collisions.

Rate/cost control: Monitor token consumption. If ChatGPT plan limits approached (see [20†L353-L361]), schedule long jobs in off-hours or use mini model. Pin model to Codex-Mini for small tasks to save budget [18].

7. CI/CD Integration (Examples)

Concrete GitHub Actions templates help cement these ideas.

GitHub Actions YAML: Example for two Codex jobs (adjusted for brevity):

```
name: Multi-Agent Codex CI
on: [pull_request]
permissions:
  contents: read
jobs:
  codex-scaffold:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - name: Setup Codex CLI
        uses: openai/setup-codex@v1
        with:
          version: 5.3.0
      - name: Run Scaffold Agent
        env:
          CODEX_HOME: ${{ runner.temp }}/codex_scaffold
          CODEX_API_KEY: ${{ secrets.CODEX_API_KEY }}
        run: |
          codex exec --json --profile scaffold --output-last-message
scaffold.md \
            "Scaffold new endpoints for feature X in backend and frontend"
      - name: Upload Scaffold log
        uses: actions/upload-artifact@v3
        with:
          name: scaffold-log
          path: codex_scaffold/log.json
  codex-review:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3
      - uses: openai/setup-codex@v1
        with: { version: 5.3.0 }
      - name: Run Reviewer Agent
        env:
          CODEX_HOME: ${{ runner.temp }}/codex_review
          CODEX_API_KEY: ${{ secrets.CODEX_API_KEY }}
        run: |
          codex exec --json --profile reviewer --output-last-message
review.txt \
            "Review changes on this pull request and suggest fixes if needed"
      - name: Upload Review log
        uses: actions/upload-artifact@v3
        with:
          name: review-log
          path: codex_review/log.json
```

Key points: each job has its own CODEX_HOME, logs output, and does not auto-commit changes (no `--apply`). Instead, outputs are saved (we can use the CLI's `--output-last-message` to capture

suggestions in a file). A follow-up step could comment on the PR using this text (using GitHub Action or a small script), but **only after human review**.

Safe Auto-Fix Workflow: If you allow one agent to auto-fix (e.g. `codex exec --apply`), do so on a feature branch or fork, not on `main`. Better: have the agent open a PR itself with fixes, which a human then reviews. (We can script it: after running Codex with `--apply`, run `git push` to a new branch and use GitHub CLI to open a PR.)

GitLab CI: Similarly define parallel jobs with CI matrix and `artifacts`. The config syntax differs but concept is identical (use Docker or `openai/setup-codex` if available).

8. Local Dev Workflows

- **Project CODEX_HOME:** Encourage each developer to set `CODEX_HOME` inside project (e.g. `$PWD/.codex`) via `.env` file or launcher script. This isolates history per project.
- **Multiple CODEX_HOME:** You can manually switch CODEX_HOME in terminal or via scripts. Example shell script `run-codex-agent.sh`:

```
#!/bin/bash
CODEX_HOME="$PWD/.codex_$1" codex exec --profile $1 "$2"
```

- **Profiles:** Use `--profile` as in CI, to apply role-specific settings. Profiles are defined in `.codex/config.toml`.
- **Pre-commit Hooks:** Use pre-commit to run Codex checks (in *read-only* mode) before code commits. For example, a hook `codex-diff` could run `codex exec --no-apply --profile fixer "Check modified files for lint errors"` and fail on errors. This ensures issues caught early.
- **VS Code Integration:** The Codex VSCode extension can also use multiple agents (via worktrees/profiles), but keep settings consistent with CLI configs. Consider using Workspace settings to specify which skills or prompts the extension should use by default.

9. Testing & Review Strategy

- **Generated Code Tests:** Any code generated by Codex should have unit tests. E.g. if Codex adds a function, write tests in the same run or immediately after. Use `--output-schema` to have Codex generate JSON for API, then validate with schema tests.
- **Contract Testing:** For APIs, have OpenAPI schemas; use codex to generate initial spec, but verify with `openapi-generator` and real requests in CI.
- **Human Review:** Do **not** blindly commit Codex output. Always gate with a PR. Even if an agent suggests fixes, a developer must review diffs. Enforce this in policy.
- **Avoid Ephemeral Commits:** Disable auto-apply on mainline. If using `--apply`, do so in a side branch or local dev only.

10. Governance & Policy Checklist

- **Auto-commit vs PR:** Generally, restrict auto-commit. Only allow an agent to `--apply` changes in a draft branch. Most changes should require a PR with a human reviewer.

- **Code Ownership:** If agents generate code, clearly annotate generated sections (e.g. comments `// Generated by Codex CLI`). Assign ownership of each agent's output to a team member for accountability.
- **Secrets Rotation:** Rotate Codex API keys periodically (every few months) and immediately if a breach is suspected.
- **Artifact Retention:** Keep logs/artifacts for X months (depending on policy). Remove older ones to save space but keep enough for audits.
- **Incident Response:** If a Codex agent behaves maliciously (e.g. injected a bad patch), treat it like a supply-chain incident: revert commits, revoke tokens, check logs for intrusion. Codex tasks should be reproducible from logs (see next section).
- **Review Policies:** Require 2nd-party sign-off on changes from Codex if code impacts security/data. Maintain an ADR if adopting major agentic changes.
- **Compliance Note:** For PCI/GDPR, Codex has no PCI certification. If in-scope data is involved, do not send raw sensitive data to Codex prompts. Use placeholders or hashed tokens.

11. Migration & Rollback

If generated code causes regressions or security issues:

- **Revert:** Immediately revert the offending commit (e.g. `git revert`).
- **Investigate:** Check the Codex logs to identify prompts/inputs that led to it. Use the runId/commit to find JSON events. This is why metadata logging is crucial.
- **Reproduce:** To safely reproduce the run, pin down all inputs: same CLI version, model, AGENTS.md and rules version. Use JSONL and prompt as a *replay* (you may have to remove `--apply`).
- **Lockfiles:** For reproducibility, commit a lockfile for AGENTS.md or config (e.g. a script that regenerates them deterministically). Consider storing the entire prompt in a versioned spec.
- **Rollback Plan:** If large refactors, use the *Strangler Pattern* [16]: introduce new code around old code, switch over piece by piece. Codex can help generate adapters (anti-corruption layer) if needed.
- **Dual-write Pattern:** If splitting services in future, prepare adapters: run feature flags. Keep DB migrations backward-compatible while rewriting logic in new service.
- **Undo Generated Code:** If an agent made many file changes, and you must undo, use `codex exec --help` (hypothetically) or rely on `git revert`. No built-in "undo" yet – for now manual reverts.

12. Decision Criteria for Single vs Multiple Instances

Decide *when* to introduce multiple instances. Table 2 provides signals.

Table 2. *When to stick with one agent vs add more agents.*

Signals & Criteria	Continue Single Agent	Scale to Multiple Agents
Team size	1 developer (solo)	3+ developers (multiple domains)

Signals & Criteria	Continue Single Agent	Scale to Multiple Agents
Task complexity	Simple, single-domain tasks	Distinct domains (frontend/backend) [1]
Workflow bottleneck	All tasks fit in one session/agent	Long-running runs, tasks queuing up [3]
Code reviews backlog	Low (manual suffice)	High (introduce Codex agent for reviews)
Security requirements	Low (internal tools sufficient)	High (external agentic scans for code) [2]
Budget for Codex	Very limited (avoid extra runs)	Adequate for parallel runs (monitor usage) [19]
Maintainability risk	One set of agent instructions to update	Many roles to keep in sync (higher effort)

Moving from one to many is **high effort** (setting up orchestration, skills, rules) but can yield **high benefit** if workflow demands parallelism. Be cautious: issue [4] shows Codex stability can suffer under parallel load [3], so ensure you use separate processes and not one interactive session with background jobs.

13. Prioritised Next Steps

1. **Establish baseline:** Confirm a single Codex workflow (e.g. one agent scaffolding feature) works end-to-end with your tech stack.
2. **Set up** `AGENTS.md`**:** Draft project instructions (npm test rules, style guides, SQL lints) so all agents share context [8].
3. **Define roles & profiles:** Write a simple `.codex/config.toml` with profiles (`scaffold`, `fixer`, `reviewer`) and sandbox rules. (Use defaults from examples above.)
4. **Implement basic rules:** Create `.codex/rules/default.yaml` forbidding destructive commands (`rm -rf`, `sudo`) and requiring approval for `workspace-write` actions.
5. **CI pipeline skeleton:** Add a GitHub Actions job that runs `codex exec` (profile `scaffold`) with `--json` on PR. Validate it runs without errors.
6. **Logging:** Ensure the CI job archives the JSON log. Also make developers log CLI output locally (e.g. redirect to file).
7. **Secret management:** Store your Codex API key in GitHub Secrets. In local dev, use environment variable (`CODEX_API_KEY`).
8. **Pre-commit hook:** Integrate a pre-commit rule to run lint/ruff/tests, or even a read-only Codex lint check on changed files.
9. **Review process:** Draft a policy: all Codex changes require PRs. Train the team on how to review Codex-generated diffs.
10. **Monitor usage:** Keep an eye on Codex usage dashboard [20], and set alerts for approaching limits. If needed, switch some tasks to GPT-5.1-Codex-Mini [18].

Following these steps builds a foundation. The full report below expands each area with detailed guidance, templates, and references.

References

- OpenAI Codex CLI docs (Configuration, Rules, AGENTS, Skills, Non-interactive, GitHub Action) [21] [8] [22].
- OpenAI blog: *"Unrolling the Codex agent loop"* [23].
- GitHub Issues (Codex CLI): #3280 (multi-agent orchestration request) [1]; #10887 (parallel tasks bug) [3].
- Rezvani (2025): on multi-agent CI for security pipelines [2].
- GitHub Actions Security (Least Privilege, Secrets) [12] [24].
- OWASP Cheat Sheet: Secrets Management (least privilege, rotation) [25].
- SLSA provenance spec (build metadata) [16] [17].
- Trew Knowledge News (Feb 2026): multi-agent orchestration context [26].
- OpenAI Codex Pricing & usage limits [27].

---

[1] Orchestration of multiple agents · Issue #3280 · openai/codex · GitHub
https://github.com/openai/codex/issues/3280

[2] Codex CLI - Complete Guide from Zero to Agents | Medium
https://alirezarezvani.medium.com/openai-codex-cli-from-broken-install-to-multi-agent-orchestration-production-guide-07d8b7d513ef

[3] Codex 5.3: Parallel Long-Running Tasks Can Cause Session Crashes / Truncated Output · Issue #10887 · openai/codex · GitHub
https://github.com/openai/codex/issues/10887

[4] [8] Custom instructions with AGENTS.md
https://developers.openai.com/codex/guides/agents-md/

[5] [6] [7] [9] [21] Command line options
https://developers.openai.com/codex/cli/reference/

[10] [22] Agent Skills
https://developers.openai.com/codex/skills/

[11] [13] [14] [25] Secrets Management - OWASP Cheat Sheet Series
https://cheatsheetseries.owasp.org/cheatsheets/Secrets_Management_Cheat_Sheet.html

[12] [15] [24] Secure use reference - GitHub Docs
https://docs.github.com/en/actions/reference/security/secure-use

[16] [17] SLSA • Provenance
https://slsa.dev/spec/v1.0/provenance

[18] [19] [20] [27] Codex Pricing
https://developers.openai.com/codex/pricing/

[23] Unrolling the Codex agent loop | OpenAI
https://openai.com/index/unrolling-the-codex-agent-loop/

[26] AI This Week: Multi-Agent Orchestration Becomes Reality - Trew Knowledge Inc.
https://trewknowledge.com/2026/02/06/ai-this-week-multi-agent-orchestration-becomes-reality/