Codex vs Claude Code

Hands-On Test Protocol — Three Reproducible Tasks

A practical, first-party benchmark for comparing OpenAI Codex and Claude Code across three workflow types — bug fixing, multi-file refactoring, and greenfield development. Run the identical prompt on both tools, reset the repo between runs, and score the results on four dimensions: correctness, idiomatic style, test coverage, and maintainability.

1. One-Time Setup

Claude Code

Native installer is recommended; npm also works (needs Node 18+):

```
npm install -g @anthropic-ai/claude-code
claude --version
```

First launch opens the browser for OAuth (token stored in ~/.claude/), or set ANTHROPIC_API_KEY. Requires a Pro / Max / Team / Enterprise / Console account. On Windows, use WSL for the CLI.

Codex CLI

```
npm install -g @openai/codex     # or: brew install --cask codex
codex --version
```

On launch choose “Sign in with ChatGPT” (bundled, no extra cost) or set OPENAI_API_KEY. The sandbox blocks network access by default.

Record before you start: exact model used (Claude Code: /model; Codex: shown in the TUI), your plan/tier, the date, OS, and Node version.

2. Repo Setup & Reset Discipline

One git repo per task. Commit a clean baseline so you can reset to it before every run.

```
mkdir codex-vs-claude && cd codex-vs-claude
mkdir task1-bugfix task2-refactor task3-greenfield

# inside each task folder, after adding starter files:
git init && git add -A && git commit -m "base"

# greenfield is empty, so force an empty baseline:
cd task3-greenfield && git commit --allow-empty -m "base"
```

Reset to the clean baseline before each run (handles new files the agent adds):

```
git reset --hard HEAD && git clean -fd
```

Save each tool's output to a labelled branch so you can diff them later, e.g. `git commit -m "codex-result" && git branch codex-task1`.

3. The Run Loop (identical for all three tasks)

- Run each tool in a separate session so neither sees the other's work.
- Reset to baseline, launch the tool, confirm the model, then paste the prompt verbatim.
- Keep both in approval mode (Claude Code “ask” / Codex “Auto”) for a fair comparison.

- Intervention rule: only respond if the agent asks a direct question or requests approval. Count each one.
- When finished, capture results: `git diff --stat` and run the tests.

```
cd task1-bugfix
git reset --hard HEAD && git clean -fd     # clean baseline
claude        # or: codex
# ...paste the task prompt, let it run...
git diff --stat
pytest -q
```

4. Task 1 — Bug Fix With Clear Reproduction

Workflow tested: bug triage. A planted logic bug counts Saturday as a business day. Watch whether each tool fixes the logic or just hacks numbers to pass.

workdays.py

```
from datetime import date, timedelta

def business_days_between(start: date, end: date) -> int:
    """Count business days (Mon-Fri) from start (inclusive) to end (exclusive)."""
    if end <= start:
        return 0
    days = 0
    current = start
    while current < end:
        if current.weekday() <= 5:   # planted bug
            days += 1
        current += timedelta(days=1)
    return days
```

test_workdays.py

```
from datetime import date
from workdays import business_days_between

def test_full_week():
    # Mon Jun 1 -> Mon Jun 8, 2026 = Mon-Fri = 5
    assert business_days_between(date(2026, 6, 1), date(2026, 6, 8)) == 5

def test_same_day():
    assert business_days_between(date(2026, 6, 1), date(2026, 6, 1)) == 0

def test_weekend_span():
    # Fri Jun 5 -> Mon Jun 8, 2026 = only Friday = 1
    assert business_days_between(date(2026, 6, 5), date(2026, 6, 8)) == 1
```

Prompt (paste byte-for-byte):

> *Two tests in test_workdays.py are failing. Find and fix the bug in workdays.py. Do not modify the test file.*

5. Task 2 — Multi-File Refactor Preserving Behavior

Workflow tested: large-scale refactoring. Three route handlers duplicate JSON validation. Score whether the shared decorator fits cleanly or feels bolted on, and whether behavior is preserved exactly.

app.py

```
from flask import Flask, request, jsonify
app = Flask(__name__)

@app.route("/users", methods=["POST"])
def create_user():
    data = request.get_json(silent=True)
    if data is None:
        return jsonify({"error": "invalid json"}), 400
    if "name" not in data or not isinstance(data["name"], str):
        return jsonify({"error": "name required"}), 400
    if "email" not in data or "@" not in data["email"]:
        return jsonify({"error": "valid email required"}), 400
    return jsonify({"created": data["name"]}), 201

@app.route("/products", methods=["POST"])
def create_product():
    data = request.get_json(silent=True)
    if data is None:
        return jsonify({"error": "invalid json"}), 400
    if "title" not in data or not isinstance(data["title"], str):
        return jsonify({"error": "title required"}), 400
    if "price" not in data or not isinstance(data["price"], (int, float)):
        return jsonify({"error": "valid price required"}), 400
    return jsonify({"created": data["title"]}), 201

@app.route("/orders", methods=["POST"])
def create_order():
    data = request.get_json(silent=True)
    if data is None:
        return jsonify({"error": "invalid json"}), 400
    if "user_id" not in data or not isinstance(data["user_id"], int):
        return jsonify({"error": "user_id required"}), 400
    if "items" not in data or not isinstance(data["items"], list):
        return jsonify({"error": "items required"}), 400
    return jsonify({"created": data["user_id"]}), 201
```

Prompt (paste byte-for-byte):

> *Refactor app.py so the repeated JSON validation logic is replaced by a single reusable decorator that accepts a spec of required fields and their expected types. Every endpoint's behavior, status codes, and error messages must stay byte-for-byte identical. Add a test_app.py that proves all three endpoints return the same responses for both valid and invalid input.*

Note: Flask must be available. Since Codex's sandbox blocks network, pre-install `flask pytest` in the folder before launching — and apply the same pre-install to both tools to stay fair.

6. Task 3 — Greenfield Feature With Tests

Workflow tested: greenfield development. Start from an empty repo so neither tool has context to lean on. Check that tests actually mock the network rather than hitting real URLs.

Prompt (paste byte-for-byte):

> *Build a Python CLI tool called linkcheck. It takes a path to a Markdown file, extracts all http(s) links, sends a HEAD request to each (3-second timeout), and prints a table of URL, HTTP status, and OK/BROKEN. Requirements: argparse interface, graceful handling of timeouts and connection errors, unit tests that mock the network (no real requests in tests), and a README with usage examples. Only requests and pytest as dependencies.*

7. How to Measure

Objective metrics (after each run)

```
git diff base codex-task1 --stat   # files / lines changed
git checkout codex-task1 && pytest -q   # first-pass test result
git diff base codex-task1          # full diff (screenshot for proof)
```

Observational metrics come from the session, not git: agent iterations (test-run → edit → re-run cycles), human interventions (approvals / questions answered), and wall-clock time. With ChatGPT-account auth, record Codex cost as “included in plan” rather than a per-task figure.

Logging table

Task	Tool	First-pass tests	Iterations	Interventions	Wall-clock	Files/lines
1	Claude Code					
1	Codex CLI					
2	Claude Code					
2	Codex CLI					
2	Codex Cloud					n/a
3	Claude Code					
3	Codex CLI					

Quality scorecard (1–5 per run)

- **Correctness** — 5: solves it cleanly, all tests pass, no regressions; 3: works with a rough edge or missed case; 1: doesn't solve it.
- **Idiomatic style** — 5: matches the file's conventions, reads naturally; 3: correct but generic; 1: awkward or fights the codebase.
- **Test coverage** — 5: meaningful tests incl. edge cases; 3: happy-path only; 1: none or trivial.
- **Maintainability** — 5: approve the PR as-is; 3: approve after minor changes; 1: needs a rewrite.

Disclose in the write-up: exact model/tier, test date, and that results are n=2–3 hands-on runs rather than a statistical benchmark. Screenshot each final diff — the diffs are the proof that turns this into real first-party testing.