

技術発表 ・ 2026.06

AI 同士の連携で "利用枠"と品質を両取りする

Claude × Codex — 自作スキルの運用と、
公式プラグインとの比較

`claude delegate → codex exec` ■

// — アジェンダ

きょう話すこと

- 01 課題 — Claude の 5 時間枠が足りない
- 02 作ったもの — 委譲スキルと安全弁
- 03 量の効果 — トークン削減の実測
- 04 質の事例 — 別 AI が拾った、Claude の見落とし
- 05 公式スキルとの比較 — 公式にも同等スキルはあるが、engine は同じ
- 06 まとめ — 量の問題で作ったツールが、質の問題を解いた

Claude の作業を、別の AI と分担する

01 仕組みを作った

Claude Code から Codex CLI を呼び、作業を別 AI に委譲する自作スキルを運用。

02 量が減った

レビューを Codex に委譲し、Claude 側消費トークンを削減。
5時間ごとの利用枠を直接温存できた。

03 質も上がった

別システムのモデルが、Claude が見落としたバグや指摘の誤りを拾う。
セカンドオピニオンとして機能した。

01

課題

Claude の 5 時間枠が足りない

01 — 前提 — 何を / どう作っているか

社内向け業務システムをClaude Codeで作っている

開発手法は **BMAD メソッド** — AI 駆動開発のフレームワーク。要件 / テスト / 実装 / レビュー を AI が読める形に構造化して回す。下の 4 段階で AI が主役になる。

① story (要件) 設計

AC (受け入れ基準) を切り出し、構造化 spec に落とす。

② RED テスト

AC を fail する test として先に書き、"正解" をテストで固定する。

③ 実装

Claude Code が spec / RED を読みながら、GREEN まで持っていく。

④ BMAD レビュー

3 層の敵対的レビュー (Blind Hunter / Edge Case Hunter / Acceptance Auditor) を diff に適用する。

01 — 背景 & 着眼点

Claude の利用枠が厳しい 一方、Codex 側は余裕あり

- Claude / Anthropic

枠: ひっ迫

5時間ごとの利用枠がボトルネック。並列で実装を進めていると 2~3 時間で上限に引っかかる。特に重いのがコードレビュー — 敵対的な 3 層レビューが PR diff を 3 並列で読む。

3.4M tokens — フルレビュー 1 回の Claude 側消費 (ccusage 実測)

- Codex / OpenAI

枠: 余裕

別 billing で並走。利用枠はまだ薄く使っており、余裕があった。

→ 判断は Claude に残し、量のある定型作業を枠の余った Codex に逃がす。

01 — 着想 & 委譲の方針

判断は Claude、量のある作業は Codex

委譲できる部分を検討し、3つのスキルを作成。

- **Claude に残す(判断)**

- Story 設計、AC 整理、RED tests の起票
- レビュー findings の triage
(採用 / defer / 棄却)
- マージ判断、PR コメント執筆

- **Codex に委譲(量のある作業)**

- レビュー本体 (diff を読んで指摘を列挙する)
- 既存パターンを増やす実装、機械的 refactor
- 修正パッチの起こし

STEP 1 ・ 実装**codex-impl**

実装本体を Codex に任せる。

STEP 2 ・ レビュー**codex-bmad-review**

3 層の敵対的レビューを委譲。

STEP 3 ・ レビュー修正**codex-fix-review**

レビュー指摘に対する修正パッチを委譲。
(使用頻度が低いので今回は割愛)

02

作ったもの

実装・レビューの委譲スキルと安全弁

実装委譲の骨格 — codex-impl

① Claude

Job 定義を作成
goal / allowed ·
forbidden paths
/ AC

→

② Wrapper

run.sh で Codex
起動
caps は job で
override 可

→

③ Codex

実装 → 編集
--patch-only で
diff だけでも可

→

④ Wrapper

Guard 判定
caps / allow-
list 違反
→ guard_failed
で停止

→

⑤ Claude

accept → verify
commit · push

専有 = Jira · branch · PR · merge · deploy は Claude。Codex は bounded worker に徹する。

委譲を暴走させない5つのガード

01 allow-list / forbidden

触っていい場所と絶対に触らない場所（.git / workflows / secrets）を明記&事後 enforce。

02 Guard caps

max_changed_files / max_diff_lines / timeout
を job 単位で指定。超えたら停止。

03 patch-only mode

適用前に diff だけ生成 → worktree を自動 revert。
人が見てから accept（実装 lane）。

04 Artifact 永続化

全 attempt の prompt / stdout / patch を保存。
「なぜこの指摘か」を後で audit できる。

05 Sandbox

Codex は限定 sandbox で動作。想定外のネットワーク呼出しやリポジトリ外への書込みを抑止する。

触ってよい範囲を、job ごとに動かせるようにした

- 最初の形 Guard caps を全 job 共通で固定。変更ファイル数 ≤ 12 / 差分行数 ≤ 1200 をハードコードしていた。
- 詰まった 大型 refactor では超過が普通。さらに **lock files / migration journal** などの自動生成ファイルが **file-count** に乗り、guard が誤検知して停止。安全側に倒すと通らず、緩めると小タスクで暴走しうる。
- 直した job spec に ## Guard caps block を置けば**その job だけ上限を上書き**（ship 済）。uv.lock / package-lock.json / _journal.json 等は file-count から**自動除外**。

抽象テーマ 安全弁は固定値ではなく、タスクごとに動的に動かす。

レビュー委譲の骨格 — codex-bmad-review

① Claude

PR と SPEC_FILE を指定
RED tests 済 committed
diff が対象

→

② Wrapper

diff を snapshot + cap
隔離 cwd / read-only
sandbox

→

③ Codex

3 層を codex exec で
並列実行 → findings
blind / edge /
auditor.md

→

④ Claude

findings を triage
採用 / defer / 棄却 →
story に追記

read-only = patch は生成しない。重いレビューの token を Claude から外すのが狙い。

3 層の敵対的レビューを並列起動

// LAYER 1

Blind Hunter

隔離した tmp cwd から実行
(プロジェクトを見ない)。
先入観なしでバグを探す。

// LAYER 2

Edge Case Hunter

境界値・例外系・想定外入
力にフォーカスして diff を
読む。

// LAYER 3

Acceptance Auditor

spec / story file との突合。
AC 違反・spec 逸脱・未実
装・矛盾を検出。

個別 layer が落ちても残りの出力は使える (failed layers を列挙、warning のみ surface)。

03

量の効果

トークン削減の実測

03 — 効果 — トークン削減（実測）

レビューを委譲すると、 Claude 側が 2.67× 軽くなる

6 PR 累計・Claude 側消費	フルレビュー	Codex 委譲
トークン	30.3M	11.3M
コスト	\$27.31	\$6.12 +Codex

2.67×

token 削減 (cost では 4.46×)

ccusage 実測 / 6 PR を codex-bmad-review と bmad-code-review で二重計測。Codex CLI 側の OpenAI 課金は別枠で、この数字は Claude 側に発生した分のみ。

04

質の事例

別 AI が拾った、Claude の見落とし

04 — 事例 ① — 主観 vs 一次資料

Claude が主観で「修正」した正しい実装を、Codex レビューが差し戻した

```
←!— 元の実装（実は正しかった） →  
<div class="error error-recoverable">...</div>  
  
←!— Claude が主観で "修正" →  
←!— "rightmost class が勝つから並び順を逆に" →  
<div class="error-recoverable error">...</div>  x  
  
/* CSS 仕様 — Codex が直接引いた */  
.error { color: red; } /* 後勝ち */  
.error-recoverable { color: orange; } /* 並び無関係 */
```

- 起きたこと

Claude が主観で「**rightmost の class が勝つ（HTML の並び順で決まる）**」と思い込み、正しかった実装を「修正」してしまった。

- そのまま Claude がレビューしていたら

一般論だが誤り。同じモデルがレビューしていたら、自分の修正を正として見落としていた可能性が高い。

- Codex レビューが拾えた理由

Codex が「**equal-specificity なら source order で決まる。class の並びは無関係**」と CSS 仕様を直接引いて反証 → revert。

抽象テーマ 主観 vs 一次資料 — 別システムのモデルだから、自分の思い込みに縛られず仕様を引いて検証できる。

04 — 事例 ② — Spec ↔ 実装の literal drift

同じ AI が書いた spec と test の名前が、ふるまい OK のままズレる

```
# story.md (Claude が書いた AC)
AC-3: 関数 test_rejects_when_quota_exceeded を実装し、
      quota 超過時に DomainError を raise すること

# 同じ Claude session が書いた test
def test_quota_limit_reached():      # ← 名前が drift
  request_with_quota_over()
  assert raises(DomainError)        # ← ふるまいは AC 通り

# CI 緑  / # Codex Auditor の指摘 ↓
AC-3 違反:
  期待 (spec): test_rejects_when_quota_exceeded
  実装 (diff): test_quota_limit_reached
  → spec 文言と test 関数名が literal 不一致
```

• 起きたこと

同じ session が AC を書き、続けて test を書く。ふるまいは AC 通りだが関数名が**"意味的に同じ"レベルで drift**。CI は緑で、Claude 自身は気づかない。

• なぜ起きる

単一 AI は自分の "意味的に同じ" drift を許容することがある。fixture / spec / test を全部同じ AI が書くと、それが閉ループに。

• 別 AI が拾えた理由

Codex Auditor は spec / story file を渡され、意味ではなく文字 literal で照合する。意味のゆらぎを汲まないなので drift を 100% 拾う。

抽象テーマ 単一 AI は自分の "意味的に同じ" drift を検出できない — 機械的な literal 照合をする別 AI が必要。

05

公式スキルとの比較

公式にも同等スキルはある。中身はどう違うか

05 — 比較 ① — engine は同じ

公式にも同等スキルはある。でも engine は同じ

OpenAI 公式の `openai/codex-plugin-cc` (v1.0.4) も、似たようなコマンド群を提供する。
だが動作の原理は共通で、異なるのは `codex` を呼び出すラッパーの部分のみ。

- `/codex:review`
- `/codex:adversarial-review`
- `/codex:rescue`
- `/codex:status`
- `/codex:result`
- `/codex:cancel`

- 公式 `codex-plugin-cc`

- 自作スキル



SAME ENGINE

codex exec

公式には無い、自作側の利点

役割を持った3層レビュー

単なる「レビューして」ではなく、Blind Hunter / Edge Case Hunter / Acceptance Auditor と役割を分けた3層を回せる。価値の中心。

guard 層

allow-list / 保護パスのブロック / 変更ファイル数・差分行数の上限。公式の汎用ラッパーには無い。

patch-only

適用前に差分だけ生成、見ってから accept。リスクの高い job も残骸ゼロ。

借りるのは UX、守るのは安全弁

PHASE ① — 安い / 高価値 / リスク ~0

非同期実行 + status / result

重いレビューを投げてもセッションが固まらない。run_in_background + ~30 行の status スクリプトで実現できる。

PHASE ② — 中価値 / 注意点あり

session resume

追い質問を diff の再送なしで回せる
(--background / --resume / --fresh)。
read-only レビューに限定し、resume 回数
cap で枠浪費を防ぐ。

どちらも手元の Codex CLI が既に持つ機能 → 薄い追加スクリプトで自前化できる。

06

まとめ

量の問題で作ったツールが、質の問題を解いた

06 — オチ

Claude の利用枠が倍になった

×2

5 時間ごとの利用 token 上限

5 月上旬、Anthropic 側で 5 時間枠が実質倍増。並列度を上げても枠をあまり心配せずに済むようになった。出発点の課題は薄まった。

では、もう要らない? → 答えは **No**。コード品質の面で十分に利用価値がある。

量の問題で作ったツールが、
質の問題を解決してくれた。

- ▶ 別の AI を "もう一人のレビュアー"
として開発フローに常設する価値
は、コスト関係なく残る。

付録

本編から外した補足 — 実装委譲の規模別ボーダー、PR 規模での使い分け、注意点、並列維持の効果。

規模別の「委譲ボーダー」が見えた

規模	判断
30 行以下 + 1 file	直接（Claude が書く） — job spec 起票コストで逆転
30 ~ 100 行	内容次第 — 機械的なら委譲、cross-cutting な設計判断なら直接
100 ~ 200 行	基本委譲
200+ 行 / 10+ files / 機械的 refactor	委譲（本領発揮ゾーン）

ざっくり、3pt 以上の構造化タスクは問答無用で委譲。それ以下は Claude 直書きで十分。

「常に委譲」ではなく、PR 規模で切り替える			
PR の性質	運用	Claude 側 cost / PR	
軽い / cosmetic / pattern 流用 (<100 行)	Codex 単独 — 95% カバー	~\$0.5-1	
中規模 auth / IDOR / write path (200+ 行)	両系統 併用	~\$5-10	
大規模 scope 拡張 / 新規設計 (1000+ 行)	併用 + scope drift gate	~\$10-20	

中規模以上では、Claude の Edge layer が codex 未検出の深い指摘（TOCTOU follow-up ・ no-op UPDATE regression 等）を拾う。質は双方向 — だから重要 PR は併用が効く。

Codex は、存在しない名前を捏造する

- 起きること

enum 名や関数名を hallucination で捏造することがある。

例 : `Recommendation.CONDITIONAL_PROCEED` (存在しない)

- 対策

RED tests を Claude 側で先に書く運用で、大半を ground できる。指摘は **必ず Claude が triage** してから採用する。

セカンドオピニオンも盲信しない。"もう一人のレビュアー"であって、決裁者ではない。