OSIIX PUBLIC DRAFT V0.15

AI補助開発を安全に進めるための運用設計ガイド

AIの速度を、実務で扱える変更・検証・記録へ変換するための作業設計。

VERSION

Public Draft v0.15

CREATED

2026-08-01

UPDATED

2026-08-01

AUTHOR

mars70

SUPPORT

AI-assisted editing

VALIDATION

GPT-5.5 / Codex CLI v0.154.6

OVERVIEW

この資料について

この資料は、私がAIを使って開発や運用補助を進める中で、迷走や手戻りを減らすために試してきた方法をまとめたものです。一般規格や唯一の正解を示すものではなく、AIを現場で活用する方法の一例として紹介しています。

私の環境では、AIに任せる範囲、変更してよい範囲、確認した事実、次に進む作業をあらかじめ整理しておくと、後から状態を追いやすくなりました。

> **目的:** AIを使った開発作業を、人間が確認・修正・再開できる形で進めるための、実践的な運用例を示すこと。

公式一次ソース: 本資料の公開版は OSIIX Library に掲載されたものを一次ソースとします。再配布物や類似資料を参照する場合は、公開元が OSIIX Library であることを確認してください。

作成・検証環境

本資料は、mars70 の個人の実務経験に基づく観測結果をまとめたものであり、モデル内部や技術的なバックグラウンドの解析を経たものではありません。作成にあたっては、2026-08-01時点の OpenAI GPT系ツール（ChatGPT GPT-5.5 Thinking / Codex CLI v0.154.6 等）による構成・文章整理・レイアウト・プロンプト例の検討支援を受けています。

本文中の承認テンプレート、VUN報告、Codex向けプロンプト例は、2026-08-01時点の GPT-5.5 Thinking / Codex CLI v0.154.6 を中心に検討した運用パターンを公開向けに整理したものです。ただし、特定モデル・将来バージョン・他社ツールでの常時再現性を保証するものではありません。AI関連製品は更新が速いため、Created / Updated の日付も併記しています。

バージョン表記について: 本資料で紹介する、変更範囲の明確化、確認結果の記録、本番操作の分離といった考え方は、特定のモデルだけを前提としたものではありません。一方、プロンプト例や運用上の細部は、モデルやツールの仕様によって調整が必要です。記載したバージョンは、本資料を作成・確認した時点の参考情報です。

この資料で紹介する名称について: R0〜R5、NOW.md、HANDOFF_PROMPT.md、Verified / Unverified / Next などは、筆者の個人環境で使っている分類・名称の例です。一般規格ではありません。考え方だけを取り入れ、名称や段階数、ファイル構成は利用環境に合わせて変更してください。

免責と利用上の注意

本資料は、法律、セキュリティ監査、認証取得、インシデント対応、または本番環境作業の専門助言ではありません。実際のシステム変更、デプロイ、DNS、メール、認証、ファイアウォール、個人情報、顧客データに関わる作業は、各組織、個人、またはグループ（集団）の責任範囲、契約、法令、承認手続きに従って実施してください。

転載・引用について

CONTEXT

1. なぜこの運用が必要なのか

AIは、短い説明からコードのたたき台を作る、ログを分類する、ドキュメントの下書きを作る、といった作業では非常に速く動きます。一方で、既存リポジトリ、運用中のサーバー、複数の設定ファイル、過去の判断、公開済みドキュメントが絡むと、単に「いい感じに直して」と頼むだけでは危険です。

問題は、AIが必ず間違うことではありません。問題は、作業範囲が広いまま進むと、どの情報が正本で、どのファイルを変更してよく、何を検証したのかが曖昧になることです。曖昧なまま会話を続けるほど、説明は増えるのに成果物が進まない状態に入りやすくなります。

避けたい進め方

目的、対象ファイル、禁止範囲、検証条件を決めずに、長い会話の中でAIに修正を重ねさせる。

目指す進め方

作業を小さく切り、Gitを正本にし、変更範囲と検証条件を固定してからAIに渡す。

目的を1つに絞る

02

変更範囲を決める

03

正本を確認する

04

検証する

05

記録して閉じる

AUDIENCE

2. 対象読者と対象外

対象読者

- ChatGPTやCodexなどのAI補助ツールを使い、既存のリポジトリを扱っている人
- 小規模チームや個人開発で、Git管理と文書管理を両立したい人
- AIの出力と実装、検証、記録のずれに困った経験がある人

対象外

- AIに開発を完全自動化させる方法
- 特定サービスのセキュリティ保証
- 法務、監査、認証取得の代替
- 本番環境への変更手順そのもの

- AI活用を、一時的な小技ではなく継続可能な作業方法として整理したい人

ここで紹介するのは、AIを現場で活用するために私が試してきた方法の一例です。必要な部分だけを、それぞれの環境に合わせて利用してください。

PRINCIPLES

3. AI補助開発の6原則

私の環境では、次の6つを意識すると、AIの出力を実際の成果物につなげやすくなりました。すべてを同じ形で導入する必要はありませんが、承認、変更範囲、正本、検証、記録をつなげて考える点は役立ちました。

01

小さい承認単位

一度に任せる範囲を、人間が確認できる大きさに切る。

02

明示的な write set

今回編集してよいファイルを事前に決める。

03

Git 正本

私の環境では、チャットではなくGitHubのremote mainとローカル作業ツリーを基準にする。

04

05

06

docs 同期

実装と同じ単位で、必要なドキュメントも更新する。

検証ログ

何を確認したか、何を確認していないかを記録する。

同時 closeout

変更、検証、文書、commit、次の作業を同じ区切りで閉じる。

BOUNDARY

4. 小さい承認単位

AIに大きな目的をそのまま渡すと、調査、実装、検証、ドキュメント更新、公開作業が一つの会話に混ざります。混ざった作業はレビューしにくく、失敗したときに戻しにくくなります。

そこで、承認は「今回やること」と「今回やらないこと」が明確になる大きさに分けます。たとえば、read-only 調査、docs-only 更新、実装、runtime 変更、公開作業は、それぞれ別の承認に分けます。

実務上の目安: 1つの承認で、変更対象、検証方法、完了条件を一目で説明できない場合、その作業は大きすぎます。

悪い例

```
この機能を公開できるところまで全部やって。
```

良い例

```
今回は docs/NOW.md と docs/CHANGELOG.md だけを更新する。
目的は、前回の検証結果を記録し、次の SINGLE_ACTION を明確にすること。
サーバー、Nginx、DNS、mail、deploy 設定には触らない。
```

WRITE SET

5. 明示的な write set

write set は、今回の作業でAIが編集してよいファイル一覧です。これを決めずに「必要なところを直して」と頼むと、AIが善意で周辺ファイルまで編集することがあります。

write set は、単なる作業指示ではなく、安全境界です。どのファイルが変更されるべきかを先に決めることで、Git diff の確認が短くなり、想定外の副作用も見つけやすくなります。

```
Approved write set:
- docs/NOW.md
- docs/CHANGELOG.md

Do not edit:
- public/**
- AGENTS.md
- README.md
- server/runtime/DNS/mail/deploy settings
```

読む範囲 / 変更してよい範囲 / 今回触らない範囲

公開作業やサーバー関連作業では、編集対象だけでなく、読んでよい範囲と触ってはいけない範囲も分けます。

区分	意味	例
読む範囲（Read set）	確認のために読んでよい範囲	docs/NOW.md, docs/CHANGELOG.md, app/config.py
変更してよい範囲（Write set）	今回編集してよい範囲	docs/NOW.md, docs/CHANGELOG.md
今回触らない範囲（Forbidden set）	今回変更・操作してはいけない範囲	Nginx、DNS、メール、systemd、デプロイ設定

秘密情報として扱うもの

.env、SSH秘密鍵、APIキー、アクセストークン、パスワードを含む設定、顧客情報や個人情報などは、単なる変更禁止ではなく、内容の表示、チャットへの貼り付け、ログ出力、要約も原則として行いません。

SOURCE OF TRUTH

6. Git 正本

AIとの会話は便利ですが、正本には向きません。会話には、途中の仮説、古い前提、失敗した案、未確認の記憶が混ざるためです。

私の環境では、現在の状態を判断するときに、GitHubのremote mainと対応するローカル作業ツリーを基準にしています。作業前後には、HEAD、origin/main、remote refs/heads/main、worktreeの状態を確認します。GitLab、Bitbucket、セルフホストGit、feature branchなどを使う場合は、そのプロジェクトで合意したリモートリポジトリと基準ブランチに読み替えてください。

```
git fetch origin main
git status --branch --short
git rev-parse HEAD
git rev-parse origin/main
git ls-remote origin refs/heads/main
```

私がmainへ直接反映する作業を完了したときは、 `HEAD == origin/main == remote refs/heads/main` かつ `worktree clean` を確認しています。feature branchやpull requestを使う場合は、現在の作業ブランチと対応するリモートブランチの一致を確認します。

DOCS SYNC

7. ドキュメントを実装と一緒に更新する

コードだけを直してドキュメントを後回しにすると、次回のAIや人間が古い説明を読み、間違った作業を始めることがあります。私の環境では、実装とドキュメントのずれが、そのまま次回の前提ミスにつながりました。

一方で、ドキュメントは増やせばよいわけではありません。役割を分け、同じ内容を何度も重複させないようにしています。

文書	役割	書くべき内容
NOW.md（筆者の現在地記録用ファイル）	現在地	現在の状態、次の作業、未確認事項、禁止境界
CHANGELOG.md	履歴	確定した変更、日付、commit、検証結果
GOAL_AND_ROADMAP.md	目的と段階	ゴール、方針、完了条件、段階
HANDOFF_PROMPT.md（筆者の引継ぎ用ファイル）	引継ぎ	次の作業者が開始するための最小情報
AGENTS.md（筆者の作業規則用ファイル）	作業規則	AIやCodexが守る恒久ルール

重要なのは、説明を増やすことではなく、次の作業で迷わないだけの情報を残すことです。

> **補足:** ここで示した文書構成は、私の個人環境における一例です。重要なのはファイル名をそろえることではなく、現在地、履歴、引継ぎ、作業規則の役割を明確にすることです。

VERIFICATION

8. 検証ログ

「確認済みです」という一文だけでは、何を確認したのか分かりません。検証ログには、実行したコマンド、確認した対象、得られた結果、確認していない範囲を残します。

```
Checks run:
- git fetch origin main
- git status --branch --short
- git diff --check
- Markdown fence / HTML comment / Jinja scan
- git rev-parse HEAD
- git rev-parse origin/main

Result:
- PASS
- HEAD == origin/main
- worktree clean
```

検証ログは、AIの説明を信じるためではなく、後から人間が確認できる状態を残すためのものです。

注意（コンテキストの冗長化防止）：コマンドの全出力や膨大なログをそのままチャット上に貼り付けさせると、画面が埋まるだけでなくAIのコンテキストを圧迫し、精度の低下や会話の破綻につながります。チャット上には実行コマンド、要約、成否判定（PASS/FAIL）を中心に出し、詳細ログはローカルファイルや専用ログへ退避する運用が現実的です。

CLOSEOUT

9. 実装と文書の同時 closeout

closeout は、作業の終わりに状態を閉じる手順です。実装、検証、ドキュメント、commit、次の作業が別々のタイミングで残ると、どれが完了でどれが未完了か分からなくなります。

同時 closeout では、変更内容、変更ファイル、検証結果、未確認事項、commit、次の作業を一つのまとまりとして記録します。

```
Closeout:
- Result: PASS
- Changed files:
  - app/example.py
  - docs/NOW.md
  - docs/CHANGELOG.md
- Checks run:
  - pytest
  - git diff --check
  - parser-safety scan
- Commit: abc1234
- Final state:
  - HEAD == origin/main
  - worktree clean
- Next single action:
  - R4_PUBLICATION_READONLY_VERIFY
```

IMPROVEMENT 1

10. 作業の種類と危険度を分ける

軽い文書更新と、本番環境に影響する操作を同じ手順で扱うと、確認が過剰になったり、逆に必要な承認が不足したりします。そこで私は、作業の種類と危険度を見分けやすくするため、R0〜R5（筆者の分類例）という区分を使っています。

この分類は一般規格ではなく、私の運用例です。重要なのはR0〜R5という名前ではなく、「読むだけ」「文書だけ」「実装」「本番操作」など、性質の違う作業を区別して伝えることです。

ランク	名前	許可される作業	別承認が必要なもの
R0	Read-only / Survey	読む、棚卸しする、現状を整理する	ファイル編集、Git操作、ランタイム変更
R1	Docs-only	承認済み文書だけを更新する	コード編集、公開物変更、ランタイム変更
R2	Diff plan	実装前の変更方針、write set、検証条件を決める	実装そのもの、デプロイ、reload/restart
R3	Implementation	承認済みwrite set内で実装し、検証する	未承認ファイル、ランタイム変更、公開反映
R4	Closeout / Publication prep	検証結果、文書、commit、公開準備を閉じる	本番公開、DNS、メール、ファイアウォール操作
R5	Runtime / Deploy	明示承認された本番・ランタイム作業	承認範囲外の追加変更、横展開

この区分が役立った点

- 軽い作業に過剰な前処理をかけないため
- 危険な作業を軽い承認で進めないため
- 作業の途中で、AIが勝手に範囲を広げるのを防ぐため
- 他人へ引き継ぐときに、作業の重さを一言で伝えるため

本番操作の扱い

本番環境や外部サービスに影響する操作を、AIの判断だけで実行させません。対象、手順、影響範囲、停止条件、ロールバック方法を人間が確認し、明示的に承認した範囲で実施します。

私の現在の運用では、最終的な実行判断と操作は人間が行っています。本資料の分類例では、この種の作業を便宜上R5としています。

IMPROVEMENT 2

11. 承認テンプレートを作り、同じ形式で使う

依頼のたびに自由な文章だけで指示すると、目的、変更範囲、禁止範囲、確認方法が抜けることがあります。そこで私は、必要な項目をまとめた承認テンプレートを作り、同じ形式で使っています。

以下は私の環境で使っている一例です。項目名や構成をそのまま採用する必要はありません。目的、変更範囲、今回触らない範囲、確認方法、作業を止める条件など、各環境で必要な項目を決めて使うことが重要です。

```
WORK_ID:
WORK_RANK:
GOAL:
SOURCE_OF_TRUTH:
REQUIRED_READS:
APPROVED_WRITE_SET:
FORBIDDEN_SET:
RESTRICTED_SECRET_MATERIAL:
ALLOWED_COMMANDS:
REQUIRED_CHECKS:
STOP_CONDITION:
DOCS_TO_UPDATE:
COMPLETION_CRITERIA:
OUTPUT_FORMAT:
NEXT_SINGLE_ACTION:
```

私の環境でのテンプレート例

```
WORK_ID: R1_DOCS_ONLY_CLOSEOUT_SYNC_YYYYMMDD_01
WORK_RANK: R1_DOCS_ONLY
GOAL: 直近の検証結果を docs/NOW.md と docs/CHANGELOG.md に記録する。
SOURCE_OF_TRUTH: GitHub remote main and matching local worktree.
REQUIRED_READS:
- docs/NOW.md
- docs/CHANGELOG.md
APPROVED_WRITE_SET:
- docs/NOW.md
- docs/CHANGELOG.md
FORBIDDEN_SET:
- public/**
- server/runtime/DNS/mail/deploy settings
- unrelated files
RESTRICTED_SECRET_MATERIAL:
- .env
- SSH private keys
- API keys and access tokens
- files containing passwords, customer data, or personal data
ALLOWED_COMMANDS:
- git status / git diff / git diff --check / git rev-parse
REQUIRED_CHECKS:
- changed files exactly match approved write set
- Markdown parser-safety scan
- final HEAD == origin/main if commit/push is in scope
DOCS_TO_UPDATE:
- docs/NOW.md
- docs/CHANGELOG.md
COMPLETION_CRITERIA:
- verified facts and unverified items are recorded
- NEXT_SINGLE_ACTION is one item only
OUTPUT_FORMAT: closeout summary only
NEXT_SINGLE_ACTION: R2_DIFF_PLAN_FOR_NEXT_CHANGE
```

効果

- AIが勝手に目的を広げにくくなる
- レビュー対象が明確になる

- 同じ形式で履歴を残せる
- Codexへ渡すプロンプトとして再利用しやすい
- 作業前に、不足している承認や検証条件を見つけやすい

IMPROVEMENT 3

12. Verified / Unverified / Next に分けて記録する

AI補助開発で危険なのは、未確認の推測が、いつの間にか確認済みの事実として扱われることです。これを防ぐには、作業報告の最後に Verified、Unverified、Next を必ず分けて書きます。

項目	意味	書く内容
Verified	実際に確認したこと	実行したコマンド、確認したファイル、得られた結果
Unverified	まだ確認していないこと	runtime未確認、deploy未実施、外部サービス未確認など
Next	次に1つだけやること	次の承認単位、次の作業ID、次の検証境界

```
Verified:
- docs/NOW.md and docs/CHANGELOG.md were updated.
- Changed files match the approved write set.
- Markdown parser-safety scan passed.
- HEAD == origin/main after push.

Unverified:
- Runtime state was not checked.
- Nginx reload/restart was not performed.
- DNS/mail/server settings were not changed.

Next:
- R2_DIFF_PLAN_FOR_PUBLICATION_ROUTE
```

分けて記録する理由

この3項目に分けると、観測した事実、確認していない範囲、次の一手が分かれます。結果として、次の作業者が「何を信じてよいか」「何をまだ信じてはいけないか」を判断しやすくなります。

> Verified / Unverified / Next は、AIの説明を短くするための型ではありません。
> 確認済みの事実と未確認の推測を混ぜないための安全装置です。

PRACTICE 1

13. 失敗時の撤退とロールバック

AI補助開発では、失敗しない前提ではなく、失敗したときに安全に止める前提が必要です。Verified とした後に問題が出る場合もあるため、撤退手順を先に決めます。

状態	止める基準	安全な対応
同じエラーが続く	同種の修正を2回以上試して改善しない	会話を打ち切り、Observed facts と Unverified だけを残して再起動する
write set拡大が必要	未承認ファイルの編集が必要になる	作業を止め、追加承認を取る
検証不能	再現コマンドや期待結果が不明	実装を続けず、R2 Diff plan に戻す
危険操作が必要	reload、restart、デプロイ、DNS、メール、ファイアウォール が必要になる	R5として別承認に切り出す
誤変更が見つかる	承認外ファイル、秘密情報、不要な大規模変更が混ざる	差分を保存してから、破棄・revert・再設計のどれにするか人間が決める

ロールバックの注意

`git reset --hard` は強力ですが、未保存の変更を消します。公開資料では「安全な撤退手順」として、まず diff の保存、変更ファイルの確認、ブランチや作業コピーの扱いを明示し、破棄操作は人間の承認後に限定するのが安全です。

```
失敗時プロトコル:
1. 追加実装を止める
2. git status と git diff --stat で変更範囲を確認する
3. 必要なら diff を保存する
4. Verified / Unverified / Risks を記録する
5. 破棄・revert・再設計のどれにするか人間が決める
6. HANDOFF_PROMPT.md または closeout 形式で再起動する
```

PRACTICE 2

14. セッションとコンテキスト管理

会話が長くなると、古い指示、失敗した修正案、途中の仮説が混ざり、AIが現在の承認境界を見失いやすくなります。1つの承認単位が終わったら、必要に応じて新しいセッションへ切り替える前提で運用します。

リセット基準	理由	再起動時に渡すもの
1つの承認単位が完了した	完了済みの文脈を次作業へ持ち越さない	最新Git状態、closeout、Next single action
同じ説明を繰り返し始めた	会話が前進していない可能性が高い	Observed facts、現在のwrite set、未解決エラー
AIが禁止範囲を提案した	承認境界を見失っている可能性がある	FORBIDDEN_SETを含む短い承認テンプレート
エラー修正がループした	失敗案がコンテキストに残っている	再現条件、最新差分、失敗した試行の要約
チャットが重い/長い	コストと精度の両面で不利	HANDOFF_PROMPT.mdを使った最小再開情報

```
セッション再起動用の最小情報:
- Source of truth: GitHub remote main + local worktree
- Current HEAD / origin/main
- Approved WORK_RANK and write set
- Verified facts only
- Unverified items
- Next single action
- Forbidden set
```

PRACTICE 3

15. 機密情報漏えい防止の自動化

「秘密情報を書かない」という注意だけでは不十分です。人間もAIもミスをする前提で、誤コミットや公開前の混入を検出する仕組みを取り入れています。

対策	目的	運用上の注意
.gitignoreの明確化	.env、秘密鍵、生成物、ローカル設定をGit対象外にする	サンプルは .env.example などに置き、実値は入れない
pre-commitフック	commit前に形式、秘密情報、不要ファイルを検査する	導入手順と失敗時の対応を文書化する
secret scanning	APIキー、トークン、秘密鍵らしき文字列を検出する	誤検知もあるため、例外ルールを管理する
CIでの検査	ローカルの見落としをpull requestやmain前に検出する	失敗ログに秘密情報を出さない
公開前スキャン	HTML/PDF/READMEに内部名や秘密情報が残っていないか確認する	公開版と内部レビュー用を分離する

公開版での書き方

具体的な実環境パス、IPアドレス、内部ホスト名、顧客名、実トークン風の値は例にも使わない。例は sample、example、placeholder に統一します。

PRACTICE 4

16. JSON / YAML 承認テンプレート

私のようにテンプレートをツールやスクリプトでも再利用したい場合は、Markdownだけでなく JSONやYAMLにしておく方法もあります。これは必要な人向けの具体例であり、通常のチャット利用ではMarkdownや文章形式だけでも十分です。

> **キーの役割:** `forbidden_set` は、今回の作業で変更・操作してはいけない対象を示します。`restricted_secret_material` は、変更禁止に加えて、内容の参照、表示、要約、ログ出力、チャットへの貼り付けも避ける対象を示します。
> Text版、YAML版、JSON版で同じ役割になるようにそろえています。

YAML版

```yaml
work_id: R3_IMPLEMENTATION_EXAMPLE_YYYYMMDD_01
work_rank: R3_IMPLEMENTATION
goal: Implement the approved small change only.
source_of_truth:
  - GitHub remote main
  - Corresponding local worktree
approved_write_set:
  - app/example.py
  - tests/test_example.py
forbidden_set:
  - server/runtime/DNS/mail/deploy settings
  - unrelated files
restricted_secret_material:
  - .env
  - SSH private keys
  - API keys and access tokens
  - files containing passwords, customer data, or personal data
required_checks:
  - pytest
  - git diff --check
  - changed files match approved write set
```

JSON版

```
{
  "work_id": "R3_IMPLEMENTATION_EXAMPLE_YYYYMMDD_01",
  "work_rank": "R3_IMPLEMENTATION",
  "goal": "Implement the approved small change only.",
  "source_of_truth": ["GitHub remote main", "Corresponding local
worktree"],
  "approved_write_set": ["app/example.py", "tests/test_example.py"],
  "forbidden_set": ["server/runtime/DNS/mail/deploy settings", "unrelated
files"],
  "restricted_secret_material": [".env", "SSH private keys", "API keys
and access tokens", "files containing passwords, customer data, or
personal data"],
  "required_checks": ["pytest", "git diff --check", "write set
verification"],
  "closeout_format": ["Result", "Changed files", "Verified",
"Unverified", "Risks", "Next"]
}
```

> JSON/YAMLはAIの解釈を助けますが、承認そのものではありません。人間が承認した範囲を機械可読にするための形式として扱います。

PRACTICE 5

17. レビュー粒度とアンチパターン

承認単位を小さくすることは有効ですが、小さすぎると人間のレビュー待ちがボトルネックになります。安全性と進捗のバランスを取るため、適切な粒度を決めます。次の15-30分という時間は、私が確認しやすいと感じている目安であり、一般的な基準ではありません。

判定軸	適切な粒度	小さすぎる/大きすぎる兆候
レビュー時間	1承認または1PRを15-30分程度で確認できる	毎回数分の承認が多発する / 1回で全体レビューになる
目的	1つの目的に集中している	複数機能、リファクタ、文書整理が混ざる
差分	変更ファイルと理由を説明できる	diffを見ても意図が追えない
検証	必要なテストや確認が明確	確認条件が曖昧、または多すぎる
影響範囲	失敗時に戻せる範囲	本番影響や横展開が暗黙に含まれる

細かくしすぎる場合の注意

承認単位を細かくしすぎると、確認と説明だけが増え、実装や文書の完成が遠のくことがあります。小さい承認単位は、作業を止めるためのものではありません。人間が確認できる範囲を保ちながら、成果物、検証、記録を着実に積み上げるための方法です。

CODEX PROMPT

18. Codexに渡すときの最小プロンプト

CodexやAI開発支援ツールへ渡すプロンプトは、長ければよいわけではありません。必要なのは、正本、承認範囲、禁止範囲、検証、停止条件、出力形式です。

```
Use the local worktree as the source of truth.
Do not infer current state from chat history.

WORK_RANK: R3_IMPLEMENTATION
GOAL: Implement the approved small change only.
APPROVED_WRITE_SET:
- app/example.py
- tests/test_example.py
- docs/NOW.md
- docs/CHANGELOG.md
FORBIDDEN_SET:
- server/runtime/DNS/mail/deploy settings
- unrelated refactors
RESTRICTED_SECRET_MATERIAL:
- .env
- SSH private keys
- API keys / access tokens
- files containing passwords or personal data
REQUIRED_CHECKS:
- tests for changed behavior
- git diff --check
- changed files exactly match approved write set
- parser-safety scan for edited Markdown
STOP_CONDITION:
- stop if write set must expand
- stop if runtime/deploy operation becomes necessary
- stop if restricted secret material must be read or displayed
CLOSEOUT_FORMAT:
- Result
- Changed files
- Verified
- Unverified
- Risks
- Next
```

NOTICE

19. 利用上の注意とライセンス

本資料は、AI補助開発を安全に進めるための実務上の考え方を整理した公開版資料です。筆者の経験に基づく観測結果であり、特定の製品、法令、セキュリティ基準への適合を保証するものではありません。

利用上の注意

実際のシステム変更、デプロイ、DNS、メール、認証、ファイアウォール、個人情報、顧客データに関わる作業は、各組織、個人、またはグループ（集団）の責任範囲、契約、法令、承認手続きに従って実施してください。

ライセンス

免責

本資料は、法的助言、セキュリティ監査、認証取得、インシデント対応、本番環境作業の専門助言ではありません。公開資料として内部情報を含まないよう汎用化していますが、各環境への適用は利用者の判断と責任で行ってください。

AI支援表記: 本資料は、mars70 の個人の実務経験に基づく観測結果をまとめたものです。作成にあたってはAIによる構成・文章整理・レイアウト支援を受けています。2026-08-01時点の GPT-5.5 Thinking / Codex CLI v0.154.6 を中心に検

討した運用パターンを含みますが、特定モデル・将来バージョンでの動作を保証するものではありません。

SUMMARY

20. まとめ

AI開発で重要なのは、AIを使うか使わないかではありません。重要なのは、AIにどこまで任せ、どこから人間が境界を引くかです。

私の環境では、小さい承認単位、明示的な write set、Git 正本、docs 同期、検証ログ、実装と文書の同時 closeout を意識することで、AIの速さを実際の成果物につなげやすくなりました。

さらに、失敗時の撤退、セッション管理、秘密情報検出、JSON/YAMLテンプレート、レビュー粒度を足すことで、運用を回し始めた後の壁にも対応しやすくなります。

これらは唯一の正解ではありませんが、AIとの作業が長くなり、状態が分からなくなったときに、どこへ戻ればよいかを示す目印として役立っています。