Contents

Completion-First Skill v1.3 Redesigned

Standalone consolidated edition for Codex / multi-agent development.

このファイルは、Completion-First v1.3 Redesigned の Skill 本体と詳細リファレンスを一つに統合したものです。

Part I —SKILL.md

Completion First

Treat the project specification as the source of truth for **what to build**. Treat this skill as the policy for **how to reach a human-evaluable build with the least useful work, context duplication, and reasoning cost**.

Objective

Optimize for:

1. **Time to First Human Feedback**
2. **Cost per Successful Completion**
3. Low duplicate work / context / retries

Do not optimize for agent count, activity volume, test count, or parallelism.

Core rules:

- Completion > Robustness > Elegance.
- Useful progress > agent activity.
- “Just in case”is not a valid implementation reason.
- Success is a STOP condition.
- Minimally hardened is acceptable; fake completion is not.
- Delegation is optional. Do not delegate when delegation overhead is likely to exceed the work saved.
- Parallelism is optional. Spawn only the minimum useful workers.

Four-layer control model

```
Completion-First
    -> Goal / Scope / Phase / Stop

Self-Organizing Worker Pool
    -> Who should work on what

Adaptive Work Packaging
    -> How large a job each assignee should receive

Adaptive Reasoning
    -> Model / reasoning budget
```

These are separate controls.

A powerful parent may identify a large `Work Item`, but it must not assume that the same item is an appropriate unit of delegation to a cheaper model.

> **Task size is relative to the assignee.**

Start of work

1. Read the specification.
2. Identify Goal, Acceptance Criteria, Scope, Out of Scope.
3. Resolve only uncertainty that materially blocks implementation.
4. Mark questions requiring a working artifact as `PROTOTYPE_REQUIRED`.
5. Freeze approved requirements.
6. Identify the shortest real end-to-end path to the First Evaluatable Build.

For detailed phase policy, read `references/Completion-First-Subagent-Workflow.md` only when needed.

Work Item vs Work Packet

Never conflate these.

Work Item

Outcome-level work on the project/critical path.

Example:

```
W-014: Make customer data persist across restart.
```

A Work Item may be too large or uncertain for direct delegation.

Work Packet

A bounded unit shaped for a specific model/tier so that it has a high probability of being completed in one fresh context.

Example:

```
P-014B
Target: Luna medium
Goal: Persist validated form data through customer_store.py
Required Context: customer_store.py + save call contract
Boundary: customer_store.py, customer_form.py
Acceptance: restart reads the saved record
Out of Scope: migration, UI redesign, DB replacement
```

The parent/Sol-class orchestrator owns packetization.

Read `references/Adaptive-Work-Packaging.md` when work is delegated to a subagent or when a packet fails because of scope/context mismatch.

Assignee-relative sizing

Choose the largest packet that the target model is likely to complete **without rediscovering the architecture, reopening product decisions, or requiring repeated retries**.

Do not make packets tiny merely because the assignee is cheaper.

Too small: - repeated briefing; - repeated file/context reads; - excessive handoffs; - merge overhead.

Too large: - broad rediscovery; - partial implementation; - retries; - reasoning escalation; - eventual return to Sol.

Aim for **high one-shot completion probability at the largest efficient packet size**.

Default tier guidance

Use these as heuristics, not rigid file-count rules.

```
Mechanical / deterministic edit
    -> Luna low/medium

Clear local implementation with narrow context
    -> Luna medium

Clear multi-file vertical slice with moderate coupling
    -> Terra medium

Direction is clear but implementation/state coupling is difficult
    -> Terra high or Luna high when the context remains local

Root cause, architecture, requirement interpretation,
cross-cutting direction, or packetization itself is unclear
    -> Sol medium/high
```

If model names differ, map them to equivalent capability tiers.

Do not hand Luna/Terra a Sol-sized problem simply because it is cheaper.

Delegation gate

Before spawning/delegating, ask:

1. Is there a concrete Work Packet rather than a vague Work Item?
2. Is the packet sized for the target model?
3. Can the target probably finish it without broad repository rediscovery?
4. Is required context smaller than simply letting the parent finish the work?
5. Will delegation reduce expected total time/token/rework?
6. Is another worker already doing materially the same work?

If the answer to 1-5 is not convincingly yes, do not delegate yet. The parent may implement directly, investigate first, or repackage.

Minimal self-organizing worker pool

Use fixed phases/gates, but allow workers to select among **eligible prepared Work Packets** rather than fixed professions.

Do not spawn a full pool by default.

```
1 worker:
one useful packet on the critical path

2 workers:
two independent critical-path packets
or one packet + one justified independent investigation

3 workers:
only when a third independent high-value packet exists
and its benefit exceeds spawn/context/merge cost
```

For small/medium work, 3 active workers is the normal cap.

Workers follow:

```
OBSERVE -> CLAIM -> WORK -> PUBLISH -> RELEASE
```

Workers claim only packets appropriate for their tier.

Reasonless duplicate work is prohibited. Parallel READ may be justified. Parallel WRITE to the same responsibility is normally prohibited.

Read `references/Self-Organizing-Worker-Pool.md` only when multiple workers are actually used.

Compact context policy

Context duplication is a cost.

Give workers only:

- current Goal;
- relevant Acceptance Criteria;
- packet boundary;
- verified facts;
- required files/contracts;
- stop/escalation rules.

Do not automatically give:

- full Discovery history;
- other workers'full transcripts;
- discarded hypotheses;
- unrelated repository context.

Share failed attempts compactly so another worker does not repeat them.

Packet failure classification

Do not automatically solve every failure by increasing reasoning.

After a failed packet, classify first:

```
PACKET_TOO_LARGE
    -> split/repackage

PACKET_TOO_SMALL
    -> merge adjacent work if repeated context dominates

MISSING_CONTEXT
    -> enrich packet, do not restart broad discovery

DIRECTION_UNCLEAR
    -> return to Sol

IMPLEMENTATION_COMPLEX
    -> consider higher reasoning / stronger implementation tier

SPEC_CONFLICT
    -> stop and escalate

OUT_OF_SCOPE
    -> defer
```

A scope/context failure is a **packaging failure**, not evidence that the worker needs more reasoning.

Adaptive reasoning

Reasoning effort is an execution budget, not a quality badge.

Default:

```
Luna/Terra medium:
normal implementation

high:
direction is clear, but implementation reasoning is genuinely difficult

Sol medium/high:
cause / architecture / requirement / direction is unclear

xhigh:
exception only after the hypothesis and direction remain well supported
and the remaining difficulty is implementation complexity
```

Do not choose high merely because the task is important.

After two unsuccessful fixes for the same failure, stop autonomous retry and escalate/repackage.

For detailed reasoning policy, use the relevant section of `references/Completion-First-Subagent-Workflow.md`.

Completion tests

Before human evaluation, test enough to prove:

- the target starts;
- the happy path works;

- the real primary processing path is connected;
- required persistence/read/external I/O is real;
- major Acceptance Criteria pass;
- no critical failure blocks evaluation.

Do not expand testing because more tests are possible.

Before adding a test:

> If it failed, would the current task or First Evaluatable Build be incomplete?

If no, defer it unless it covers a critical risk.

No facade completion

Do not claim end-to-end completion through undisclosed:

- mocks;
- stubs;
- fakes;
- hard-coded success values;
- skipped persistence;
- fallback data hiding failure.

Disclose any intentional mock/fake and why it does not invalidate human evaluation.

First Evaluatable Build Gate

Require:

- target starts;
- main flow works end to end;
- real primary path is connected;
- required persistence/I/O is real;
- major Acceptance Criteria pass;
- critical blockers are absent;
- startup instructions were actually verified;
- evaluator data exists;
- Evaluator Quick-Start exists;
- a human can begin meaningful evaluation in roughly five minutes;
- mocks/known limitations/deferred work are disclosed.

When this gate passes:

> **STOP the worker pool.**

Remaining nice-to-haves are not a reason to continue.

Independent review

Reviewers are not normal workers.

Give them a compact Review Packet, not the full worker conversation:

- approved requirements / Acceptance Criteria;
- relevant diff/final files;
- test/execution evidence;
- known limitations;
- Evaluator Quick-Start.

Before human evaluation, a reviewer may block only for CRITICAL/REQUIRED issues, not style, speculative architecture, optional hardening, or nice-to-have tests.

Phase reset

After the First Evaluatable Build:

1. Create a compact Completion Snapshot.
2. Prefer fresh context for the next phase.
3. Pass requirements, current code, snapshot, human feedback, and only necessary design facts.
4. Do not drag full implementation/reviewer transcripts forward.

Optional telemetry

When comparing workflow versions or tuning packet sizes, use lightweight event/metrics logging from `references/Benchmark-Telemetry.md`.

Do **not** enable verbose telemetry by default during normal development. Measurement itself must not become material overhead.

Final check

Before continuing any work, ask:

- Does this directly improve the probability of reaching the next human-evaluable state?
- Is the current worker the right capability tier?
- Is the packet too large, too small, or appropriately sized for that worker?
- Would direct parent execution be cheaper than delegation?
- Is reasoning escalation actually needed, or is this a packaging/direction problem?
- Is duplicate context/work being created?
- Has success already occurred?

If success has occurred, stop.

Part II —Completion-First Subagent Workflow

Completion-First Subagent Workflow

cc-sdd 向け「まず評価可能な完成」を最優先する開発ワークフロー

Version: 1.3 Redesigned
Target: cc-sdd v3 / Codex・Claude Code 等の subagent 対応環境
Primary Goal: **Time to First Human Feedback（人間が実際に触って評価し、方向性のフィードバックを返せるまでの時間）を最小化する**

Cost Goal: **そこへ到達するまでの総トークン消費・重複作業・不要な Context 転送を最小化する**

Delegation Goal: **Work Item を担当モデルに合わせた Work Packet へ成形し、再調査・再試行・handoff を含む総コストを最小化する**

> **重要度はプロジェクト基準。作業粒度は担当能力基準。推論予算は問題の難しさ基準。**

Secondary Goal: **Time to First Evaluatable Build（人間が実際に触って評価できる最初のビルドまでの時間）を最小化する**

---

0. このワークフローの目的

このワークフローは、AI エージェントが以下のような「局所的には正しいが、全体として完成を遅らせる行動」に陥ることを防ぐ。

- 「念のため」の防御実装を増やし続ける
- 重要度の低いテストを繰り返す
- 実装途中で大規模リファクタリングを始める
- 将来要件を先回りして抽象化する
- Reviewer と Implementer が細部を巡って往復する
- 同じ失敗に対して修正を繰り返す
- Scope 外の問題を見つけ、その場で修正し始める
- テストを通すこと自体が目的になる
- 人間がまだ一度も触っていないのに品質だけを高め続ける

このワークフローでは、最初の開発目標を「最終品質」ではなく次の状態に置く。

> **要求された主要動作が、本来の実処理経路を End-to-End で一通り通過し、人間が最小限の準備だけで実際に起動・操作し、「これでよい／違う」を判断できる状態。**

これを **First Evaluatable Build / 第一評価可能版** と呼ぶ。

第一評価可能版は「見た目だけ動くデモ」ではない。

- UI だけ動いて裏側が未接続
- 本来の保存先ではなく一時メモリだけを使う
- 本来の API 呼び出しを固定値で置き換える
- エラー時にダミーデータへ fallback して成功したように見せる
- モック・stub・fake の利用を隠す

といった状態は、原則として第一評価可能版とは認めない。

> **ロバストでなくてもよい。だがハリボテであってはいけない。**

第一評価可能版の完成後に、人間からのフィードバックを反映し、その後に本格的な網羅テスト、堅牢化、リファクタリングを行う。

---

1. 最上位原則

すべての親エージェント・サブエージェントは以下の優先順位に従う。

1. **Requirements への適合**
2. **主要機能の End-to-End 完成**
3. **人間が実際に評価できる状態への到達**
4. **現在の Acceptance Criteria を証明する最小限のテスト**
5. **重大な既知不具合の除去**
6. 堅牢性
7. 保守性
8. コードの美しさ
9. 将来拡張性
10. Nice-to-have

特に以下を強制する。

> **Completion > Robustness > Elegance**
>
> **「念のため」は実装理由にならない。**
>
> **評価可能とは、見た目ではなく本来の主要処理経路が End-to-End で接続されていることを意味する。**
>
> **モック・stub・fake・固定値・fallback で成功を偽装してはならない。使用する場合は明示する。**
>
> **Success is a STOP condition. 要求を満たしたら止まる。**

---

2. フェーズ構成

```
USER
  ↓
Discovery + Grill
  ↓
Requirements
  ↓
REQUIREMENTS FREEZE
  ↓
Minimal Design
  ↓
Vertical-Slice Tasks
  ↓
Guided Self-Organizing Worker Pool
  ↓
Implementation / Task-local Minimal Verification
  ↓
Integration / Runnable Build
  ↓
Evaluator Quick-Start 生成
  ↓
FIRST EVALUATABLE BUILD GATE
  ↓
Stage A Completion Snapshot
  ↓
PHASE CONTEXT RESET
  ↓
Human Evaluation
  ↓
Required Fixes
  ↓
Full Validation
  ↓
Hardening
  ↓
Refactor / Nice-to-have
```

重要なのは、**Human Evaluation を可能な限り前へ持ってくること**である。

---

3. Discovery + Grill

目的

実装開始後に仕様を再解釈しなくて済む程度まで、重要な意思決定を前倒しする。

kiro-discovery に grill-me 的な質問能力を組み込む場合、このフェーズのみ探索的でよい。

積極的に確認するもの

- ユーザーが達成したい最終目的
- 必須機能
- 明確な非目標
- 主なユーザーフロー
- 入力と出力
- データ保存の有無

- 失敗時に最低限必要な挙動
- 外部サービス・依存関係
- 対応環境
- 「実物を触らないと決められない部分」
- Acceptance Criteria
- 既存仕様との衝突
- Scope

Discovery で許可される行動

- 前提を疑う
- 別案を提示する
- 矛盾を指摘する
- 決定木を掘る
- Scope の分割を提案する
- 「これは今決められない」と判定する

Discovery の終了条件

以下をすべて満たしたら終了する。

- 実装開始を妨げる重大な未決事項がない
- 主要な Acceptance Criteria を記述できる
- Scope / Out of Scope が分離されている
- 実物を見なければ決められない事項が識別されている

重要

「もっと詳しく決められる」は Discovery 継続理由にならない。

人間が実物を触らないと判断できないものは、無理に会話で確定しない。

```
PROTOTYPE_REQUIRED:
- 対象:
- 実際に見て判断する内容:
- 現時点の仮決定:
```

として後段へ送る。

---

4. Requirements Freeze

Requirements が人間に承認された時点で **REQUIREMENTS FREEZE** を宣言する。

以後、Implementer / Tester / Reviewer / Debugger は Requirements を勝手に変更してはならない。

許可されること

- Requirements 通りに実装する
- 曖昧さを報告する
- 矛盾を報告する
- 実装不能条件を報告する

禁止されること

- 「一般的にはこうだから」で仕様を補完する
- 「こちらの方が便利だから」で仕様を拡張する

- 将来性を理由に仕様を追加する
- Reviewer の好みで要件を変更する
- テスト都合で仕様を変更する

Requirements 変更が必要な場合は必ず親へ返す。

```
SPEC_ESCALATION:
- 該当 Requirement:
- 問題:
- なぜ現在の仕様のままでは進めないか:
- 最小の変更案:
- 変更しない場合の影響:
```

---

5. Minimal Design

Design Agent の目的は「理想的なアーキテクチャ」ではない。

> **承認済み Requirements を完成させるために必要十分な設計を作る。**

Design 優先順位

1. 実装可能
2. 既存コードとの整合
3. タスク分割可能
4. End-to-End で早く動く
5. 必要最低限の保守性

第一評価可能版まで原則禁止

- 未要求の Plugin Architecture
- 汎用 Framework 化
- 将来機能向け抽象化
- 「今後増えるかもしれない」型への対応
- 不要な Repository / Service 層
- 大規模な既存コード再構成
- 目的のない Dependency 更新

例外は「それをしないと Requirements を実装できない」場合のみ。

---

6. Task Planning —Vertical Slice First

タスクは可能な限り **Vertical Slice** で作る。

避ける分割

```
1. DB を全部作る
2. API を全部作る
3. UI を全部作る
4. テストを全部作る
```

この形式では途中段階を人間が評価しづらい。

推奨

```
Slice 1:
最小入力
→ 処理
→ 結果表示

Slice 2:
保存
→ 再読込
→ 表示

Slice 3:
編集
→ 保存
→ 反映

Slice 4:
主要エラー処理
```

各 Slice は可能なら、それ単独で何らかのユーザー価値を確認できる形にする。

各タスクに必須の項目

```
Goal:
このタスクでユーザー視点の何が可能になるか

Acceptance Criteria:
このタスクが完了したと判断する条件

Boundary:
変更してよい範囲

Out of Scope:
今回変更しないもの

Required Tests:
Acceptance Criteria を証明する最小限のテスト

Stop Conditions:
作業を止める条件

Escalation Conditions:
親へ返す条件
```

---

7. Implementer Policy

Implementer は「独自に製品を改善する設計者」ではない。

役割は、

> **承認済み Task Brief を、必要最小限の変更で完成させる実装担当。**

Implementer MUST

- 現在の Task Goal に集中する
- Acceptance Criteria を満たす

- Boundary 内だけ変更する
- 既存仕様を尊重する
- 必要最低限のテストを実行する
- Scope 外の発見事項は記録して先へ進む
- 成功条件を満たしたら終了する

Implementer MUST NOT

- Scope 外のバグをついでに修正する
- 大規模リファクタリングを行う
- 将来要件を実装する
- 未要求の fallback を追加する
- 未要求の retry を追加する
- 未要求の logging を大量追加する
- 「念のため」validation を追加する
- Dependency を理由なく更新する
- Acceptance Test を書き換えて成功させる
- 本来の処理をモック・stub・fake・固定値で置き換えたまま完成扱いする
- fallback で失敗を隠して成功扱いする
- モック利用を報告せずに End-to-End 完成と主張する
- 別タスクのコードまで綺麗にする
- Requirement を再解釈する

「気づいた改善」の処理

実装しない。

```
DEFERRED:
- 内容:
- 発見理由:
- 現在の Task に不要な理由:
- 将来対応する場合の候補:
```

---

8. Test Policy —二段階化

テストを以下の 2 段階に分離する。

Stage A: Completion Tests

第一評価可能版までに行う。

目的:

> **現在の Requirements と主要ユーザーフローが動くことを証明する。**

対象:

- 起動確認
- Happy Path
- Acceptance Criteria に直結するテスト
- 主要な統合確認
- 明白なクラッシュ
- データ破壊など重大事故につながる最低限の異常系

原則対象外:

- 網羅的境界値
- Rare edge case

- exhaustive test
- fuzzing
- 微小な性能改善
- 全環境互換性
- 「念のため」の大量回帰テスト

End-to-End Smoke Check の最低要件

第一評価可能版では、少なくとも主要ユーザーフローについて以下を確認する。

- UI または入力点から開始できる
- 本来の主要ロジックを通る
- 本来のデータ保存・取得経路を通る
- 本来の外部 I/O が Requirement 上必須なら、その経路を通る
- 最終的なユーザー可視結果まで到達する
- 再起動・再読込が主要フローの一部なら、保存結果が実際に残っている
- モック・stub・fake を使う場合、その箇所と理由が明示されている

以下は Smoke Check 成功とみなさない。

UI 表示だけ成功
固定ダミーデータを返して成功
保存処理をスキップして成功
失敗時に `fallback` 値を返して成功
本番経路を通らずモックだけで成功

> **Smoke Check は厳密にする。ただし網羅的にはしない。**

TDD を使う場合

RED テストは現在の Acceptance Criteria を直接証明するものに限定する。

> **テストを増やせることと、今増やすべきことを混同しない。**

Stage B: Full Validation

Human Evaluation 後に行う。

対象:

- Regression
- Edge cases
- Error handling
- Integration
- Full suite
- 対応環境確認
- 必要な性能試験
- セキュリティ上必要な検証
- 障害復旧
- 長期運用上重要なケース

---

9. Reviewer Policy

Reviewer の目的はコードを「自分好みに改善する」ことではない。

第一評価可能版までは **Completion Reviewer** として振る舞う。

Reviewer が Required として指摘してよいもの

- Requirement 違反
- Acceptance Criteria 未達
- 明確な機能不全
- Task Boundary 違反
- 重大な回帰
- データ破壊
- 起動不能
- 人間評価を阻害する問題
- 明確なセキュリティ事故につながる問題

Required にしてはいけないもの

- コードスタイルの好み
- より美しい抽象化
- 将来拡張性
- 「こちらの方が一般的」
- 軽微な重複
- 現 Requirements と無関係な改善
- Nice-to-have

指摘には必ず Severity を付ける。

```
CRITICAL:
```
完成・安全・データ整合性を阻害する。必須修正。

```
REQUIRED:
Acceptance Criteria または Requirements に違反。必須修正。

DEFERRED:
```
改善価値はあるが第一評価可能版には不要。修正禁止。

Reviewer の原則

> **Reviewer は「さらに良くできる理由」ではなく「今このタスクを Reject すべき理由」を証明する。**

Reject 理由を証明できなければ PASS する。

---

10. Debugger Policy

Debugger は「いろいろ試して直す」担当ではない。

役割は **Root Cause の特定**。

デバッグ原則

> **One hypothesis → One minimal change → One verification**

複数箇所を同時に変更してはいけない。

Circuit Breaker

同一問題について、

1. 仮説 A → 修正 → 失敗
2. 仮説 B → 修正 → 失敗

まで行ったら、3 回目の修正を自動実行しない。

以下を返す。

```
DEBUG_ESCALATION:
- 現象:
- 試した仮説:
- 得られた証拠:
- 現時点の Root Cause 候補:
- 次に確認すべき一点:
- Scope 拡大が必要か:
```

禁止

- 原因不明のまま変更範囲を広げる
- Dependency をまとめて更新する
- キャッシュ削除等を無根拠に繰り返す
- 「環境依存だろう」と証拠なく決めつける
- テストを弱めて通す

---

11. Loop Prevention Rules

11.1 Fix Ping-Pong

以下の状態を検出したら停止する。

```
A を直す
→B が壊れる
→B を直す
→A が再び壊れる
```

同じ失敗集合が 2 巡したら設計または Root Cause 問題として Escalate。

11.2 Reviewer Ping-Pong

Reviewer Reject が同一タスクで 2 回続いたら Debugger または親へ移す。

Reviewer と Implementer だけで無限往復しない。

11.3 Refactor Spiral

現在の Task Goal 達成前のリファクタは原則禁止。

許可条件:

- 現構造では Acceptance Criteria を満たせない
- 変更しないと重大な回帰が避けられない

それ以外は Deferred。

11.4 Test Expansion Loop

テスト追加前に毎回問う。

> このテストが失敗した場合、現在の Task を未完成と判定するか？

NO なら第一評価可能版前には追加しない。

11.5 Dependency Upgrade Loop

Dependency 変更は以下のいずれかの場合のみ。

- Requirements で指定
- 現バージョンでは実装不能
- 既知の重大脆弱性への対応が今回必須
- 明確な互換性問題の Root Cause である

変更する場合は 1 回に 1 依存を原則とする。

11.6 Scope Creep

新しい要求が自然発生した場合、

```
NEW_SCOPE_CANDIDATE:
- 内容:
- なぜ必要に見えるか:
- 現在の Requirements に含まれるか: YES / NO
```

NO なら実装しない。

---

12. Global STOP Conditions

全サブエージェント共通。

以下のいずれかで停止する。

SUCCESS

Acceptance Criteria を満たした。

→ **即終了。追加改善禁止。**

BLOCKED

現在の Boundary 内では進めない。

→ Escalate。

SPEC CONFLICT

Requirement 同士、または Requirement と Design が矛盾。

→ Escalate。

SCOPE EXPANSION REQUIRED

別モジュールの大幅変更、Dependency 更新、アーキテクチャ変更等が必要。

→ Escalate。

REPEATED FAILURE

同一問題への修正を 2 回行って解決しない。

→ Escalate。

OUT OF SCOPE DISCOVERY

別の問題を発見。

→ Deferred に記録して現在 Task を継続。

---

13. Evaluator Quick-Start Runbook

第一評価可能版を人間へ渡す前に、**Evaluator Quick-Start Runbook** を必ず生成する。

目的はドキュメント整備ではない。

人間が説明を読み解く時間を使わず、すぐ起動し、5 分程度で主要フローを試せる状態にする。

Runbook は原則として 1 画面〜短い 1 ページに収める。

必須内容

```
EVALUATOR QUICK START

起動:
1. ...

必要な前提:
- ...

確認用データ:
- ...

まず試すこと:
1. ...
2. ...
3. ...

今回確認してほしいこと:
- ...
- ...

既知の未対応:
- ...
- ...

モック / stub / fake:
- 使用なし
または
- 使用箇所:
```

- 理由:
- 本番経路との違い:

終了方法:
- ...

Runbook 生成時のルール

- 長い背景説明を書かない
- 設計思想を再掲しない
- 開発者向け README の代替にしない
- 人間が評価するために不要な情報を載せない
- コマンドを載せる場合は実際に検証済みのものだけを書く
- テストデータが必要なら同梱または自動生成する
- API キー等が必要なら最小手順を明記する
- 可能ならワンコマンドまたは単一 EXE 等で起動できる形を優先する

Runbook 完了条件

[ ] 起動手順を実際に検証した
[ ] 評価用データが用意されている
[ ] 主要フローを 5 分程度で試せる
[ ] 人間が今回どこを評価すべきか分かる
[ ] 既知の未対応が明示されている
[ ] モック等の利用状況が明示されている

---

14. First Evaluatable Build Gate

以下をすべて満たしたら **FIRST EVALUATABLE BUILD** と判定する。

[ ] アプリケーションまたは対象機能を起動できる
[ ] 主要ユーザーフローを End-to-End で実行できる
[ ] End-to-End が見た目だけでなく本来の主要処理経路を通っている
[ ] 本来必要な保存・読込・外部 I/O が実接続されている
[ ] モック・stub・fake・固定値・fallback の利用箇所が明示されている
[ ] 主要 Acceptance Criteria を満たす
[ ] ユーザーが実際に操作・確認できる
[ ] 評価を妨げる Critical Bug がない
[ ] 必要な Completion Tests が通っている
[ ] Evaluator Quick-Start Runbook がある
[ ] 評価用データが用意されている
[ ] 起動手順を実際に検証済み
[ ] 主要フローを 5 分程度で評価開始できる
[ ] 未対応事項が Deferred として明示されている

この段階では以下は必須ではない。

[ ] 全 edge case 対応
[ ] 完全なテストカバレッジ
[ ] 最終的なリファクタ
[ ] 完璧なログ
[ ] 将来拡張性
[ ] Nice-to-have

第一評価可能版は「雑なプロトタイプ」を意味しない。

Requirements に沿って実際に動くが、まだ最終的な磨き込みをしていない完成状態を意味する。

---

15. Stage A Completion Snapshot / Phase Context Reset

第一評価可能版の Gate を通過したら、Stage A で使っていた長い実装会話をそのまま Stage B へ持ち込まない。

> **Phase Transition は原則として Context Reset を伴う。**

目的:

- Discovery〜Task 1..N の長い履歴によるコンテキスト圧迫を避ける
- Stage A の「完成優先・Hardening 禁止」を Stage B へ誤って持ち込まない
- Stage B の Hardening 思考を次の新機能 Stage A へ持ち込まない
- 古い仮説・失敗ログ・一時的な判断を新しい Agent へ継承しない

Stage A Completion Snapshot

Phase 終了時に以下だけを圧縮して保存する。

```
PHASE HANDOFF

Approved Requirements:
- ...

Implemented:
- ...

Verified:
- ...

Major End-to-End Flow:
- ...

Known Limitations:
- ...

Deferred:
- ...

Mocks / Stubs / Fakes:
- ...

Human Evaluation Focus:
- ...

Do NOT Assume:
- ...

Current Build / Entry Point:
- ...
```

Context Reset Policy

Stage B 開始時は可能な限り fresh agent を使う。

新 Agent へ渡すもの:

- 承認済み Requirements
- 現在のコード

- Stage A Completion Snapshot
- Evaluator Quick-Start
- Human Feedback
- 必要な Design / Task 情報

原則渡さないもの:

- Discovery の全会話ログ
- 失敗した仮説の全文
- Implementer / Reviewer 間の長いやり取り
- 一時的な検討メモ
- 既に棄却された代替案

必要な決定事項は会話履歴ではなく Snapshot へ昇格させる。

---

16. Human Evaluation Gate

第一評価可能版完成後、原則として本格 Hardening へ進む前に人間の評価を挟む。

確認する。

- 操作感は想定通りか
- UI/UX の方向は正しいか
- ワークフロー自体が正しいか
- 不要な操作がないか
- 欲しかった情報が表示されるか
- 本当に必要な機能が抜けていないか
- Requirement 自体を修正すべき点が見つかったか

人間フィードバックは以下に分類する。

`REQUIRED_FIX:`
第一評価可能版の方向性・利用価値に影響する。

`SPEC_CHANGE:`
`Requirements` 更新が必要。

`DEFERRED:`
今すぐ不要。

`NICE_TO_HAVE:`
最終工程で検討。

---

17. Human Evaluation 後

17.1 Required Fix Pass

人間が Required としたものだけ修正する。

「ついで改善」を行わない。

17.2 Full Validation

ここで初めて feature 全体の網羅検証を行う。

- Requirements coverage
- Integration

- Regression
- Full test suite
- Error paths
- Compatibility
- 必要な非機能要件

17.3 Hardening

実際のリスクに基づいて行う。

- Validation
- Retry
- Fallback
- Logging
- Recovery
- Security
- Performance

「念のため」ではなく、対象リスクを明記する。

```
HARDENING_ITEM:
- Risk:
- Probability:
- Impact:
- Mitigation:
- Why now:
```

17.4 Refactor

機能完成と検証の後に行う。

目的を明記できないリファクタは禁止。

---

18. cc-sdd v3 への適用方針

cc-sdd v3 の既存構造は維持する。

```
kiro-discovery
→ kiro-spec-init
→ kiro-spec-requirements
→ kiro-spec-design
→ kiro-spec-tasks
→ kiro-impl
→ kiro-validate-impl
```

変更するのは主に **各フェーズの判断基準**。

18.1 kiro-discovery

追加する思想:

- grill-me 的な前提確認
- 決定木の重要枝を確認
- Prototype Required の識別
- Scope / Out of Scope 明示
- Discovery を長引かせるための質問は禁止

Discovery Stop Rule

```
If remaining uncertainty can only be resolved by seeing or using working software,
do not continue interrogating.
Mark it PROTOTYPE_REQUIRED and proceed toward implementation.
```

18.2 kiro-spec-requirements

Requirements 末尾に以下を追加する。

```
## Freeze Policy

Once approved, these requirements are frozen for implementation.

Implementers and reviewers MAY:
- report ambiguity
- report contradiction
- report infeasibility

They MUST NOT:
- silently reinterpret requirements
- add new product behavior
- expand scope for robustness or future-proofing

Any required change must be escalated to the parent/human.
```

18.3 kiro-spec-design

Design rules へ追加:

```
Prefer the simplest design that satisfies approved requirements.

Do not introduce abstractions for hypothetical future requirements.

Do not refactor unrelated existing architecture unless required to satisfy
the current acceptance criteria.

Optimize for the shortest path to an end-to-end evaluatable build.
```

18.4 kiro-spec-tasks

Tasks 生成ルールへ追加:

```
Prefer vertical slices over layer-by-layer decomposition.

Every task must define:
- Goal
- Acceptance Criteria
- Boundary
- Out of Scope
- Required Tests
- Stop Conditions
- Escalation Conditions
```

可能なら、最初の数タスクだけで主要ユーザーフローが触れる状態にする。

18.5 kiro-impl —Implementer Prompt

既存の task-local TDD は維持する。

ただし以下を追加する。

```
COMPLETION-FIRST POLICY

Your job is to complete the current Task Brief, not improve the whole codebase.

Priority:
1. Spec compliance
2. End-to-end task completion
3. Minimal evidence required by acceptance criteria
4. Robustness
5. Elegance

"Just in case" is not a valid implementation reason.

Do not:
- fix unrelated issues
- perform speculative refactors
- add future-proof abstractions
- broaden test scope without a task requirement
- change dependencies unless required
- modify acceptance tests merely to make them pass

If you discover useful but unnecessary work, record it as DEFERRED.

If acceptance criteria are satisfied, STOP.
Success is a stop condition.
```

18.6 kiro-review —Completion Reviewer

第一評価可能版までは Reviewer の Reject 基準を狭める。

```
Reject only for:
- requirement violation
- acceptance criteria failure
- boundary violation
- meaningful regression
- critical runtime failure
- data corruption risk
- issue that prevents human evaluation

Do not reject for:
- style preference
- speculative architecture improvement
- future extensibility
- minor duplication
- optional hardening
- nice-to-have tests

Classify every finding:
CRITICAL / REQUIRED / DEFERRED
```

```
Only CRITICAL and REQUIRED may block completion.
```

18.7 kiro-debug

既存の bounded debug 思想をさらに明示する。

```
Use:
One hypothesis → One minimal change → One verification.

After two unsuccessful fix hypotheses for the same failure:
STOP modifying code.
Return DEBUG_ESCALATION.
```

18.8 kiro-verify-completion

検証結果を二種類に分ける。

TASK VERIFIED

現在 Task の Acceptance Criteria を満たした。

FIRST EVALUATABLE BUILD VERIFIED

主要フローが人間評価可能。

この時点では Production Ready を主張しない。

18.9 kiro-validate-impl

Human Evaluation 前後で目的を分ける。

Pre-Human

最小限の Integration Smoke Check。

目的:

> 人間が評価可能か確認する。

Post-Human

従来通りの Full Validation。

- requirements coverage
- design alignment
- full suite
- integration
- regression

---

19. モデル選択・推論強度ポリシー

モデル選択と推論強度は別の制御軸として扱う。

「重要なタスクだから high」は禁止する。

推論強度を上げる理由は、現在の承認済みタスクを完成させるために、追加の推論能力が必要であることだけである。

19.1 基本ルーティング

機械的・決定的変更
→ `Luna low または medium`

通常実装
→ `Luna medium`

実装方針は明確だが、状態・依存・整合性の推論が難しい
→ `Luna high`

根本原因・設計・`Requirements` 解釈・実装方針そのものが不明確
→ `Sol medium / high`

根本原因と実装方針が十分支持され、`Luna high` でも実装上の複雑さで失敗
→ `xhigh` を例外的に検討

利用環境のモデル名や推論ラベルが異なる場合は、最も近い役割・強度へ読み替える。

19.2 Luna の標準推論強度

通常の実装作業は **Luna medium** を標準とする。

`low` は機械的かつ決定的な変更に限って使用してよい。コスト節約だけを理由に low へ下げ、ミスによる再実行を増やしてはならない。

19.3 Luna high への昇格

Luna を high へ上げるのは、**実装方針は十分明確だが、正しく実装するための推論負荷が高い場合**に限る。

対象例:

- 非同期処理
- 並行処理・競合制御
- 複雑な状態管理
- 複数コンポーネント間の依存
- 複数関数・複数モジュールにまたがる整合性修正
- UI Automation など状態依存の複雑な実装
- 複雑な PowerShell / shell 処理
- 保存・復元・キュー・Promise 等が絡む処理
- 根本原因は明確だが、実装時の整合性維持が難しい作業

単に「重要なタスクだから」という理由だけで high を選択してはならない。

19.4 Sol へ戻す条件

不明なのが「どう実装するか」ではなく、**何が原因か／何を実装すべきか**である場合、Luna の推論強度を上げ続けない。

以下の場合は Sol medium/high へ戻す。

- 根本原因が十分に確立していない

- 現在の原因仮説と観測証拠が矛盾する
- Requirements と既存実装が衝突している
- 設計判断またはプロダクト判断が必要
- 正しい実装方針自体が不明
- 複数回の失敗により現在の仮説が怪しくなった

  能力不足と仮説間違いを混同しない。

19.5 xhigh への昇格

xhigh は例外的に使用する。high の失敗直後に機械的に xhigh へ上げてはならない。

以下をすべて満たす場合にのみ検討する。

1. Sol が根本原因候補と実装方針を十分に整理している
2. 証拠によって原因仮説が現在も支持されている
3. Luna high で実装を試みた
4. 失敗結果が原因仮説そのものを大きく否定していない
5. 残る問題が主として実装整合性・状態追跡・多段推論の複雑さにある
6. 推論能力を上げることで成功率改善が期待できる具体的理由がある

原因仮説そのものが怪しくなった場合は xhigh へ進まず Sol へ戻す。

19.6 Completion-First との整合

推論強度の昇格は、現在の Task を完成させるために必要な場合のみ許可する。

以下を理由とした昇格は禁止する。

- 品質向上
- 念のため
- 将来性
- Scope 外の改善
- 追加テスト
- Reviewer の好み
- optional hardening
- リファクタリング欲求

  推論強度は品質ランクではなく、現在の問題を解くための実行予算である。

---

20. 親エージェントの役割

親は実装作業を細かく奪わない。

主な責務:

- 現在の Phase を管理
- Requirements Freeze を守る
- Task priority を守る
- Scope expansion を許可または禁止
- Escalation を裁定
- Reviewer 間の衝突を裁定
- Human Evaluation Gate を管理
- Deferred List を保持
- サブエージェントがゴールを見失っていないか監視

親が常に確認する質問

1. 今やっている作業は、第一評価可能版への到達を早めるか？
2. この作業を今やらなければ人間評価ができないか？
3. Acceptance Criteria のどれに対応しているか？
4. これは Required か、それとも単に Good Idea か？
5. 今このビルドを人間へ渡したら、5 分以内に評価を始められるか？
6. 主要フローは本来の処理経路を本当に通っているか？
7. 推論強度を上げようとしているなら、難しいのは「実装」か「原因・方針」か？

3 または 4 に答えられない作業は原則停止する。

5 が NO なら Evaluator Quick-Start または評価環境を整える。

6 が NO なら第一評価可能版として扱わない。

7 が「原因・方針」なら Luna の推論強度を上げず Sol へ戻す。
7 が「実装」で、原因・方針が十分支持されている場合のみ Luna high 等を検討する。

---

21. Reviewer を複数置く場合

複数 Reviewer は同じ観点を重複させない。

Completion Reviewer

第一評価可能版前。

見るもの:

- 動くか
- Requirement 通りか
- 人間が触れるか
- Boundary を越えていないか

Logic / Architecture Reviewer

Human Evaluation 後。

見るもの:

- 論理的欠陥
- 設計上の破綻
- failure mode
- 技術的負債
- 保守性

Usability / Simplicity Reviewer

Human Evaluation 後。

見るもの:

- 実利用で分かりにくくないか
- 過剰実装がないか
- 不要な複雑性
- ユーザー要求と挙動のズレ
- 「賢すぎる設計」になっていないか

Reviewer 同士が矛盾した場合、Implementer に判断させず親が裁定する。

---

22. Deferred Ledger

全エージェントが見つけた「良い改善案」を捨てる必要はない。

ただしその場では実装しない。

プロジェクトに一つ、Deferred Ledger を持つ。

```
## Deferred Ledger

### D-001
- Type: Refactor
- Found in: Task 3
- Description:
- Why deferred:
- User impact:
- Suggested timing:

### D-002
- Type: Edge Case
...
```

Human Evaluation 後にのみ再評価する。

---

23. 成功指標

第一評価可能版までは、コード量・テスト数を成果指標にしない。

最重要指標:

Time to First Human Feedback

> 人間が実際に起動・操作し、「正しい／違う」の最初のフィードバックを返せるまでに要した時間・工程量。

第一評価可能版を作るだけでは不十分である。

人間が以下で詰まった時間も失敗コストとして扱う。

- 起動方法が分からない
- 必要環境が不足
- テストデータがない
- API キー等の準備が不明
- 何を確認すべきか分からない

Secondary KPI:

Time to First Evaluatable Build

> 人間評価可能なビルドそのものが完成するまでに要した工程量。

補助指標:

- Requirements 承認後の仕様質問数
- Scope 外変更数
- Reviewer Reject 回数
- 同一失敗への Retry 回数

- Deferred 件数
- Human Evaluation で発覚した根本的方向違い
- Human Evaluation 前に書かれ、後で不要になったコード量
- Human Evaluation 前に書かれ、後で不要になったテスト量

特に以下を減らす。

```
Waste Before Feedback =
人間の初回評価前に作成され、
フィードバック後に不要になった実装 + テスト + 設計
```

---

24. 最小共通プロンプト

すべての実装系サブエージェントに以下を継承させる。

```
COMPLETION-FIRST GLOBAL POLICY

The immediate objective is to reach a working, human-evaluable build as early as possible.

Follow this priority:
1. Approved requirements
2. End-to-end completion
3. Human evaluability
4. Minimal required verification
5. Robustness
6. Elegance
7. Future extensibility

Rules:
- "Just in case" is not a valid reason to add code.
- Do not fix unrelated problems.
- Do not expand scope silently.
- Do not implement hypothetical future requirements.
- Do not perform speculative refactoring.
- Do not broaden tests beyond what is needed to prove the current acceptance criteria.
- Record useful non-required work as DEFERRED.
- After two failed fixes for the same problem, stop and escalate.
- If a spec change is required, stop and escalate.
- If acceptance criteria are satisfied, stop.

SUCCESS IS A STOP CONDITION.
```

---

25. 第一評価可能版前の判断ルール

迷ったら以下で判定する。

```
Q1. これをやらないと主要 Acceptance Criteria を満たせない？
 YES → やる
 NO  → Q2

Q2. これをやらないと人間が正常に評価できない？
 YES → やる
 NO  → Q3

Q3. 今放置するとデータ破壊・重大事故・重大セキュリティ問題になる？
```

```
 YES → やる
 NO  → Q4

Q4. これは本来の End-to-End 経路を通すために必要？
 YES → やる
 NO  → DEFERRED
```

---

26. このワークフローで意図的に許容するもの

第一評価可能版では、以下が多少残っていてもよい。

- 軽微なコード重複
- 一部の抽象化不足
- 非主要 edge case の未対応
- 最終的でない UI 微調整
- 最適化余地
- Nice-to-have
- 将来機能への非対応

ただし Requirements 違反・主要機能不全・重大事故リスクは許容しない。

---

27. アンチパターン早見表

症状	判定	対応
「念のため null チェックも…」	Scope 外の可能性	AC に必要か確認、不要なら Deferred
テストを次々追加	Test Expansion	現 Task を未完成と判定するテストだけ残す
実装途中で共通化	Refactor Spiral	完成後へ Deferred
Reviewer が美しさを指摘	Preference Review	Block 禁止
同じ失敗を 3 回以上修正	Debug Loop	2 回で Escalate
別モジュールのバグ発見	Scope Creep	記録のみ
「将来 ○○ になるので」	YAGNI 違反	現 Requirement に無ければ禁止
テストを変更して GREEN	Test Overfitting	原則禁止・Escalate
Dependency をまとめて更新	Upgrade Spiral	Root Cause 証明＋ 1 件ずつ
全テスト再実行を繰り返す	Verification Loop	変更影響範囲に限定
人間未評価なのに Hardening	Premature Hardening	First Evaluatable Build を先に作る
UI だけ動き裏側はダミー	Mock Fraud	本来の主要処理経路を End-to-End で通す
Phase 間で全会話を継承	Context Drag	Completion Snapshot + fresh agent
ビルド完成後に起動準備で詰まる	Evaluator Friction	Quick-Start + 評価用データ + 5 分基準
重要だから high にする	Reasoning Prestige	難しさの種類で推論強度を決める
high 失敗後すぐ xhigh	Blind Escalation	原因仮説を再評価。怪しければ Sol へ戻す
原因不明のまま Luna を高推論化	Brute-force Reasoning	Sol で原因・方針を再整理

---

28. Guided Self-Organization / Token Efficiency

v1.3 では Implementation Phase 内の Worker 割当を固定職種ではなく、Guided Self-Organization で運用できる。

詳細規約は `references/Self-Organizing-Worker-Pool.md` を参照する。

ただし最優先は自己組織化ではなく、**Completion-First + Token Efficiency** である。

28.1 Minimum Viable Swarm

- Worker は必要最小数から開始
- 小規模・中規模では通常最大 3 Worker
- OPEN な独立 Critical Path がなければ Worker を増やさない
- 新 Worker は Parallelism Gate を通過した場合だけ起動

28.2 Work Board

Worker は Compact Work Board から最重要の未担当作業を Claim する。

Board は現在状態だけを保持し、長い推論ログを共有しない。

28.3 重複制御

- 理由のない重複は禁止
- Independent Verification は許可
- Parallel READ は理由があれば許可
- Parallel WRITE は原則禁止

28.4 Context 節約

Worker には Need-to-Know Context のみ渡す。

全 Worker へ Requirements 全文、Discovery 履歴、他 Worker 会話全文を毎回配布しない。

必要な情報は親が Facts / Goal / AC / Work Board へ圧縮する。

28.5 Worker 停止

First Evaluatable Build Gate 達成時点で Worker Pool を停止する。

DEFERRED や Nice-to-have が残っていても継続しない。

28.6 Reviewer

Independent Reviewer には Worker 会話全文ではなく、Requirements / AC / Diff / Test Evidence 等の Review Packet を渡す。

これにより独立性を維持しつつ Context コストを抑える。

28.7 最終判断

並列化を判断する際は、

> **並列化で節約できる時間・作業量 > Worker 起動・Context 複製・同期・Merge コスト**

であることを要求する。

成立しない場合は Worker を増やさない。

---

29. Adaptive Work Packaging

v1.3 再設計では、Work Item と Work Packet を分離する。

`Work Item:`
プロジェクト成果基準。担当モデル非依存。

`Work Packet:`
`Target Tier` 基準。`fresh context` で一度に完遂できる可能性が高い実行単位。

親/Sol は Work Item をそのまま Luna/Terra へ渡さない。

委譲前に、

- Clarity
- Context Span
- Coupling
- Delegation overhead

を必要最小限だけ確認し、担当能力に合わせて Packet 化する。

狙うのは最小タスクではなく、

> **One-shot Completion Probability が高い最大効率 Packet**

である。

Packet 失敗時は、reasoning を上げる前に、

- PACKET_TOO_LARGE
- PACKET_TOO_SMALL
- MISSING_CONTEXT
- DIRECTION_UNCLEAR
- IMPLEMENTATION_COMPLEX
- SPEC_CONFLICT

へ分類する。

Packaging failure を Reasoning failure として扱わない。

詳細は `references/Adaptive-Work-Packaging.md` を参照する。

---

30. 最終理念

AI サブエージェントは、放置すると「仕事を見つけ続ける」。

このワークフローでは、エージェントの能力を下げるのではなく、**考えてよい範囲と終了条件を明確にする**。

Discovery では広く疑う。

Design では選択肢を絞る。

Implementation では一本道を進む。

Completion Review では完成を確認する。

そして、できるだけ早く人間に実物を渡す。

人間が実物を評価した後で、初めて再び探索範囲を広げる。

`DISCOVER:`
疑え。重要な未決事項を潰せ。

`DESIGN:`
必要十分に決めろ。

IMPLEMENT:
脇道へ行くな。完成させろ。

VERIFY:
要求を満たした証拠だけ取れ。

STOP:
成功したら止まれ。

HUMAN:
実物を見て方向を決める。

HARDEN:
方向が正しいと分かってから磨け。

まず完成させる。
必要以上の Worker・Context・推論トークンを使わずに完成させる。
完成とは「本物の主要処理経路がつながり、人間がすぐ評価できる状態」である。
ハリボテを早く作ることは Completion-First ではない。
磨くのは、その方向が正しいと確認してからでよい。

Part III —Adaptive Work Packaging

Adaptive Work Packaging

Assignee-Relative Task Sizing for Completion-First v1.3

Version: 1.3 Redesigned

---

1. 目的

安価なモデルへ仕事を渡すだけでは、トークン節約にはならない。

安価なモデルへ大きすぎる仕事を渡すと、

広範囲を再調査
→ 部分実装
→ テスト失敗
→ 再読込
→ reasoning 昇格
→ 再試行
→ 最終的に Sol へ戻る

となり、最初から上位モデルで処理するより高くつく場合がある。

逆に細かく分けすぎると、

Spawn
→ Brief
→ Read
→ 3 行修正
→ Handoff
→ 次 Worker が同じ Context を再読込

となる。

本ポリシーの目的は、

担当モデルが fresh context で一度に完遂できる可能性が高い、最大効率の Work Packet を作ること

である。

---

2. Work Item と Work Packet

Work Item

プロジェクト/ユーザー成果基準の仕事。

モデル能力とは独立して定義する。

例:

```
W-014
Goal: 再起動後も顧客情報が保持される
Priority: CRITICAL_PATH
```

Work Packet

特定の担当能力に合わせて成形した実行単位。

例:

```
P-014A
Target Tier: Terra medium
Goal: 保存と再読込を End-to-End で接続
Required Context:
- customer_store.py
- app_startup.py
- customer record schema

Boundary:
- customer_store.py
- app_startup.py

Acceptance:
- 保存
- 終了
- 再起動
- 同じ顧客が再表示

Out of Scope:
- migration
- UI redesign
- storage abstraction
```

一つの Work Item から複数 Packet が作られてよい。

複数の小さい Work Item を、Context が共通なら一つの Packet へまとめてもよい。

---

3. 誰が Packetize するか

原則として親/Sol-class orchestrator が行う。

理由:

- 全体 Goal を保持している
- Requirements Freeze を知っている
- Critical Path を把握している
- 各 Worker へ渡す Context を圧縮できる
- どこまでを下位モデルへ委譲すべきか判断できる

Worker は Packet が不適切だと判明した場合、勝手に Scope を広げず、

```
REPACKAGE_REQUEST:
- Packet:
- Problem:
- Why current boundary/context is insufficient:
- Suggested split/merge/context addition:
```

を返す。

---

4. Packet Size Principle

Packet は、

> **小さいほど良い**

ではない。

> **大きいほど良い**

でもない。

狙うのは、

> **One-shot Completion Probability が高い最大粒度**

である。

ここで One-shot とは、

- broad rediscovery を始めない
- product/design decision を再度開かない
- 大幅な Context 追加を要求しない
- 同じ問題で複数回の修正を前提としない
- Scope を勝手に広げない

状態で、Acceptance Criteria へ到達できることを指す。

---

5. 最小評価軸

Task sizing のために重いスコアリングはしない。

必要な場合だけ次の 3 軸を見る。

Clarity

```
CLEAR:
```
何を作るか、なぜ壊れているか、実装方針が概ね確定

```
UNCLEAR:
```
原因/仕様/設計/実装方針そのものが未確定

Context Span

LOCAL:
一つの責務・狭いコード範囲

MODULE:
複数の関連ファイル/同一サブシステム

CROSS-CUTTING:
複数サブシステム・設計境界・広い依存

Coupling

LOW:
状態・依存が単純

HIGH:
非同期、共有状態、永続化、複雑な依存、整合性維持が難しい

明らかなタスクではラベルを書き出す必要はない。

分類自体にトークンを浪費しない。

---

6. Tier 別の Packet 目安

Luna low

適する:

- 機械的変更
- 明確な置換
- manifest/version 更新
- 単純な定型編集

避ける:

- 原因分析
- 複数設計判断
- 広い Repository 探索

Luna medium

適する:

- 方針が明確
- 狭い Context
- 一つの責務中心
- Acceptance が明確
- 局所的な実装

典型:

```
Clarity: CLEAR
Span: LOCAL
Coupling: LOW〜中
```

Terra medium

適する:

- 複数関連ファイル
- 完結した Vertical Slice
- 既存設計に沿う実装
- Moderate integration

典型:

```
Clarity: CLEAR
Span: MODULE
Coupling: LOW〜中
```

Terra high

適する:

- 方針は確定済み
- 複数状態の整合性
- 非同期/保存/復元
- Module 内での高い Coupling

典型:

```
Clarity: CLEAR
Span: MODULE
Coupling: HIGH
```

Sol medium/high

適する:

- Root Cause 不明
- Requirement 解釈
- Design 判断
- Cross-cutting change
- Packetization
- Failed packet の再設計
- 複数 Worker 結果の統合

典型:

```
Clarity: UNCLEAR
または
Span: CROSS-CUTTING
```

Sol が重要な仕事をすべて自分で実装する必要はない。

Sol の価値は、

> **安価な Tier が一発で完遂できる形まで不確実性を減らし、仕事を成形すること**

にもある。

---

7. Terra を使う意味

Luna と Sol だけに二極化すると、

```
Luna には大きすぎる
でも Sol が直接やるほど不確実ではない
```

という中間作業が大量に発生する。

Terra は、

- 既に方針が決まっている
- しかし Luna 向けに細切れにすると handoff が増える
- ある程度の multi-file context を一度に保持した方が安い

作業をまとめるために使う。

つまり Terra は、

> **Packet を細分化しすぎないための中間 Tier**

としても価値がある。

---

8. Packet が大きすぎる兆候

以下が出たら、単純な reasoning 不足と決めつけない。

- Worker が広範囲の Repository 探索を始めた
- Design/Requirement を再検討し始めた
- Scope 外ファイルを次々読む
- 「念のため」追加調査が増える
- 主要 Goal の前に大量の前提作業が必要
- Partial completion で止まる
- Missing context が何度も発生
- 高推論へ上げないと前に進めない
- 実装より調査の方が大きくなった

対応:

```
PACKET_TOO_LARGE
→ Sol/Parent へ返す
→ 不確実性を減らす
→ Split または Target Tier 変更
```

---

9. Packet が小さすぎる兆候

- 同じ Worker/別 Worker が同じファイルを何度も読む
- 隣接 Packet ごとに同じ Brief が必要
- 変更 1〜数行ごとに Handoff
- Tests/Build を Packet ごとに繰り返す
- Merge/Review 回数が成果量に対して多い

対応:

```
PACKET_TOO_SMALL
→ Context が共通する隣接作業を Merge
→ 一つの Vertical Slice として渡す
```

---

10. Delegation vs Direct Execution

サブエージェントを使えるからといって、必ず委譲しない。

以下では親が直接処理した方が安い場合がある。

- 親が既に必要 Context をすべて読んでいる
- 作業が短い
- Worker へ渡す Brief が実作業と同程度
- 一度の小変更で完了する
- Merge/Review overhead が大きい

判断原則:

```
Expected Delegation Cost =
Spawn
+ Brief
+ Context Re-read
+ Work
+ Handoff
+ Merge
+ Retry Risk

Expected Direct Cost =
Parent additional work
```

厳密な数値計算は不要。

明らかに Delegation Cost が高ければ親が実行する。

> **Delegation is an optimization, not a requirement.**

---

11. Context Packaging

Packet には必要 Context を先回りして圧縮する。

含める:

- Goal
- Acceptance Criteria
- Boundary
- Relevant verified facts
- Required contracts/interfaces
- 関連ファイル
- Known failed attempts
- Stop/Escalation

含めない:

- 全 Discovery 会話
- 他 Worker の長い分析
- 棄却済み設計案
- Scope 外資料
- 未確認仮説

下位モデルに「必要な Context を自分で探して」と丸投げすると、価格差が Repository 再読込で消える。

---

12. Packet Failure Routing

失敗後は原因を分類する。

PACKET_TOO_LARGE

Scope/Context/責務が大きい。

→ Repackage / Split / Tier 変更。

PACKET_TOO_SMALL

Handoff/再読込が支配的。

→ Merge。

MISSING_CONTEXT

Goal は適切だが必要情報が不足。

→ 必要 Context だけ追加して再実行。

DIRECTION_UNCLEAR

Root Cause/Design/Requirement が不明。

→ Sol。

IMPLEMENTATION_COMPLEX

方向は明確で Context も適切だが実装推論が難しい。

→ high または強い実装 Tier を検討。

SPEC_CONFLICT

→ 停止して親/人間へ Escalate。

重要:

> **Packaging failure を Reasoning failure として処理しない。**

---

13. Retry Policy

同じ Packet を同じ形で何度も再試行しない。

一度失敗したら、

1. Failure class を確認
2. Packet 形状/Context/Target tier のどこが問題か判断
3. 必要なら Repackage
4. その上で再実行

Completion-First の Circuit Breaker に従い、同一問題への無意味な反復を止める。

---

14. Self-Organizing Pool との統合

Worker は Raw Work Item ではなく、**自分の Tier で実行可能とされた Prepared Packet** を Claim する。

Board 例:

```
W-014 Persist customer data

P-014A | DONE
Target: Sol
Type: Root-cause narrowing

P-014B | OPEN
Target: Terra medium
Goal: Connect save + startup reload
Write Scope: store.py, startup.py

P-014C | OPEN
Target: Luna medium
Goal: Add restart smoke fixture
Write Scope: tests/persistence_smoke.py
```

これにより自己組織化を維持しつつ、

> Luna が大きすぎる Work Item を自分で Claim して沈む

ことを防ぐ。

---

15. Adaptive Reasoning との統合

順序を守る。

1. Work Item を理解
2. 不確実性を減らす
3. Target Tier を決める
4. Target に合わせて Packetize
5. medium で開始
6. 失敗分類
7. 本当に implementation complexity なら high

high を先に選んで Packet 設計の悪さを隠さない。

---

16. Telemetry for Calibration

Packet sizing を改善する場合だけ軽量ログを取る。

推奨:

```json
{
  "work_item": "W-014",
  "packet": "P-014B",
  "target_tier": "terra-medium",
  "result": "DONE",
  "one_shot": true,
  "repackage_count": 0,
  "retry_count": 0,
  "files_read": 4,
  "files_changed": 2,
  "tests_run": 2,
  "failure_class": null
}
```

Token/credit が外部で取得できる場合は別途結合する。

見るべき指標:

- One-shot completion 率
- Repackage 率
- Retry 率
- Packet あたり Context 再読込
- Cost per Successful Completion

ログ自体を詳細化しすぎない。

---

17. 最終原則

Sol 基準で Work Item を認識してよい。

しかし、

> **Sol 基準の作業量を、そのまま Luna/Terra へ渡してはならない。**

重要度はプロジェクト基準。

作業粒度は担当能力基準。

推論予算は問題の難しさ基準。

この 3 つを混同しない。

```
What matters?
    -> Completion-First

Who should do it?
    -> Self-Organization

How much should they receive?
    -> Adaptive Work Packaging

How hard should they reason?
    -> Adaptive Reasoning
```

最終的な目的は、

> **安いモデルを使うことではない。**
>
> **最も安い総経路で、成功する完成物へ到達すること。**

Part IV —Self-Organizing Worker Pool

Self-Organizing Worker Pool

Completion-First v1.3 Redesigned Coordination Policy

Version: 1.3 Redesigned

---

1. 目的

複数サブエージェント開発で、細かな役割を事前固定せず、現在の状況から必要な役割を形成する。

ただし目的は「自己組織化そのもの」ではない。

最上位目的は次の 2 つである。

1. **Time to First Human Feedback を短縮する**
2. **そこへ到達するまでの総トークン・総作業量を削減する**

したがって、

> **エージェント数が多いほど良い**
>
> **並列度が高いほど良い**

とは考えない。

並列化による利益より、起動・説明・重複読込・同期・統合コストが大きい場合は、Worker を増やさない。

---

2. 基本構造

固定するのは Phase と Gate であり、Worker の職種ではない。

```
Parent Agent
│
├── Phase / Requirements / Scope
├── Reasoning Budget
├── Worker Count
├── Shared Work Board
│
├── Self-Organizing Worker Pool
│    ├── Worker
│    ├── Worker
│    └── Worker
│
├── FIRST EVALUATABLE BUILD GATE
│
└── Independent Review
     ├── Reviewer A
     └── Reviewer B
```

Worker はその時点で最も価値の高い**未担当 Prepared Work Packet** を選ぶ。

Raw Work Item を直接 Claim しない。親/Sol 側が担当 Tier に合わせて Packet 化してから Worker Pool へ公開する。

Reviewer は Worker Pool へ参加しない。

---

3. Minimum Viable Swarm

Worker は最初から 3 体・4 体起動しない。

必要最小数から始める。

```
Worker 1:
Critical Path が一本で明確
```

Worker 2:
独立して進められる Critical Path が 2 本ある
または
実装と独立検証を並列に行う明確な価値がある

Worker 3:
3 つ目の独立した高価値作業が存在し、
起動・共有・統合コストを上回る利益がある

小規模・中規模では原則 3 Worker を上限とする。

4 体以上は、明確な並列作業が存在し、親が追加コストに見合うと判断した場合のみ。

---

4. Parallelism Gate

新しい Worker を起動する前に、親は以下を確認する。

[ ] 明確な未担当 Prepared Work Packet がある
[ ] First Evaluatable Build の Critical Path に寄与する
[ ] 他 Worker の作業と大きく重複しない
[ ] 独立して進行可能
[ ] Worker 起動・Brief・再読込・Merge コストより期待利益が大きい

一つでも NO なら起動しない。

> **空いている計算資源は、使う理由にはならない。**

---

5. Worker の基本ループ

Worker は以下を繰り返す。

```
OBSERVE
  ↓
CLAIM
  ↓
WORK
  ↓
PUBLISH
  ↓
RELEASE
```

OBSERVE

読むものを最小化する。

- 現在の Goal
- Acceptance Criteria
- 必要な Boundary
- Work Board
- 既に確定した Facts
- 自分の作業に直接必要なコード

原則読まない。

- 他 Worker の長い会話ログ
- 全 Discovery 履歴
- Scope 外の資料

- 自分に不要な全リポジトリ調査

CLAIM

最も価値の高い未担当 Prepared Packet を選択する。

Worker は自分の Target Tier に適合しない Packet を Claim しない。

例:

```
CLAIM_REQUEST
Work Item: W-014
担当: 保存処理の原因調査
理由: Critical Path を Block しており現在未担当
Expected Output: 原因候補と再現証拠
Write Scope: none
```

Claim は短くする。

Claim 競合時は親が調停する。

WORK

Claim した範囲だけ作業する。

新しい問題を見つけても、自動的に横へ広がらない。

PUBLISH

共有するのは成果に必要な情報だけ。

```
RESULT
Work Item: W-014
Status: DONE / BLOCKED / UNRESOLVED
Verified Facts:
Artifacts / Changes:
Failed Attempts:
Remaining Blocker:
Suggested Next Work:
```

RELEASE

完了・BLOCKED・UNRESOLVED 時は Claim を解放する。

Worker が消える、Context が切れる、別作業へ移る場合も Claim を残さない。

---

6. Shared Work Board

Shared Work Board は「会話ログ」ではない。

現在の作業状態を圧縮した索引である。

推奨:

```
P-014B | CLAIMED
Parent Work Item: W-014
Priority: BLOCKER
Target Tier: Terra medium
Goal: 保存と再読込を接続
```

```
Owner: Worker-2
Write Scope: store.py, startup.py
Known Facts: 再起動後のみ消える
Blocked By: -
```

状態:

- OPEN
- CLAIMED
- BLOCKED
- UNRESOLVED
- DONE
- DEFERRED

Board には原則として「推測」を長く書かない。

共有するのは、

- 確認済み Facts
- 所有状態
- 重要な失敗
- Blocker
- 成果物位置

である。

DONE 項目は必要な結論だけ残し、詳細ログを圧縮または外へ退避する。

---

7. 重複作業

重複を全面禁止しない。

禁止するのは**理由のない重複**である。

許可例:

- 独立検証
- 異なる原因仮説
- Critical failure の再現
- 実装方式比較が意思決定に本当に必要
- 先行 Worker が BLOCKED / stale
- Reviewer による独立確認

重複する場合は理由を宣言する。

```
DUPLICATE_JUSTIFICATION:
- Existing Work:
- Why independent duplication is valuable:
- What will be kept independent:
```

---

8. Independent Investigation

独立検証は、同じ結論を増幅するために使わない。

独立 Worker へ渡すもの:

- 再現可能な事実
- 入出力
- エラー

- 関係ファイル
- Acceptance Criteria

原則渡さないもの:

- 他 Worker の未検証仮説
- 推論過程
- 「原因は X らしい」という誘導
- Reviewer の未確定見解

比較時に初めて結果を統合する。

これにより、独立性を維持しつつ、全 Context の完全複製を避ける。

---

9. Dynamic Reassignment

Worker は担当に執着しない。

変更してよい例:

- 既に他 Worker が解決
- 不要と判明
- BLOCKED
- Critical Path でなくなった
- より大きな Blocker が発生

ただし、新しいものが面白そうという理由だけで変更しない。

> **Task switching has a token cost.**

切替条件:

```
新作業の期待 Completion 寄与
>
現在作業継続の期待 Completion 寄与
+
切替・再読込コスト
```

厳密な数値計算は不要だが、この比較を意識する。

---

10. Read / Write Coordination

同じファイルを複数 Worker が読むことは許可できる。

同じファイル・同じ責務を複数 Worker が並行編集することは原則禁止。

```
Parallel READ:
理由があれば許可

Parallel WRITE:
原則 1 Worker
```

やむを得ず複数案を作る場合は、Branch / 別ファイル等で明確に隔離する。

---

11. 情報共有

共有すべきもの:

- 根本原因候補のうち証拠がある部分
- 再現条件
- 失敗したアプローチ
- 重要ファイル
- API 仕様
- テスト結果
- 変更による副作用
- Blocker
- DONE 成果物

特に失敗は共有する。

同じ失敗を別 Worker が再実行するのはトークン浪費である。

---

12. 情報共有しすぎない

全員へ全ログを配布しない。

各 Worker に渡すのは **Need-to-Know Context** だけ。

親は長い履歴を、

- Current Facts
- Current Goal
- Current Work Board
- Relevant Acceptance Criteria

へ圧縮する。

> **Context duplication is a cost.**

「念のため全員に全部読ませる」は禁止する。

---

13. 親エージェントの役割

親は職種を細かく固定しない。

管理するのは:

- Goal
- Phase
- Requirements Freeze
- Scope
- Critical Path
- Work Board
- Claim 競合
- Worker Count
- Work Item → Work Packet 成形
- Target Tier
- Reasoning Budget
- Blocker
- Integration
- First Evaluatable Build Gate

親は、

```
Worker A = 調査
Worker B = 実装
Worker C = テスト
```

とは原則指示しない。

代わりに、

```
W-014 が未担当
保存経路が現在の Blocker
```

と状況を示す。

---

14. 親が介入する条件

重複過多

理由なく同一調査・同一読込・同一修正を行っている。

作業の空白

Critical Path 上の重要 Work Item が OPEN のまま。

同調

複数 Worker が同じ未検証仮説を前提化。

局所最適

各 Worker は忙しいが Integration が進まない。

行き詰まり

同じ失敗が繰り返される。

Packet mismatch

Worker が広範囲探索・Scope 拡大・Context 不足を繰り返している。
この場合は Worker 性能不足と決めつけず、Packet Size / Target Tier / Context Package を再評価する。

Worker 過多

OPEN な独立 Packet より Active Worker の方が多い。
この場合は Worker を減らす。
介入時も可能な限り、
「未担当領域」を提示し、職種固定は避ける。

---

15. Completion-First との統合

自己組織化の評価基準は、

> 各 Worker がどれだけ仕事をしたか

ではない。

> **First Evaluatable Build へどれだけ近づいたか**

である。

優先順位:

1. First Evaluatable Build の Critical Path を直接止めているもの
2. 複数 Worker を Block しているもの
3. 次の Vertical Slice を完成させる作業
4. Completion 判定に必要な検証
5. Critical Path 上の根本原因調査
6. Non-blocking 実装
7. 改善
8. 整理・最適化

根本原因が不明でも、現在の完成に不要なら調査しない。

---

16. Adaptive Reasoning との統合

Worker は仕事を自己選択できる。

ただし Model / reasoning budget を自由に上げない。

通常 Worker:
Luna medium

実装方針明確・実装推論が複雑:
Luna high を親へ要求可能

原因・設計・Requirements 自体が不明:
Sol へ Escalate

原因十分支持 + high でも実装複雑性で失敗:
xhigh を例外検討

> **Worker は仕事を選ぶ。計算資源は親が統制する。**

これにより自己組織化が高コスト化するのを防ぐ。

---

17. BLOCKED / UNRESOLVED

Give-Up を失敗扱いしない。

新しい情報が得られないまま粘る方がコストが高い。

```
STATUS: BLOCKED / UNRESOLVED

Tried:
- ...

Verified:
```

```
- ...

Failed:
- ...

Missing:
- ...

Best Next Action:
- ...
```

同一問題への修正は Completion-First の Circuit Breaker に従う。

---

18. Independent Review

Reviewer は Worker Pool から分離する。

原則 2 Reviewer を独立に使う場合でも、**全 Worker 会話を読ませない**。

レビュー入力は圧縮する。

```
REVIEW PACKET:
- Approved Requirements
- Acceptance Criteria
- Relevant Diff / Final Files
- Test / Execution Evidence
- Known Limitations
- Evaluator Quick-Start
```

Reviewer 同士も最初のレビュー終了までは相互結論を共有しない。

これにより独立性とトークン効率を両立する。

Framework が各 Task で Reviewer を必須にする場合、Task Review は狭い Completion Check に限定し、重い二重レビューは First Evaluatable Build 候補で行う。

---

19. Pool Stop Rule

以下を満たしたら Worker Pool を停止する。

```
FIRST EVALUATABLE BUILD GATE = PASS
```

停止後に、

- OPEN な改善
- DEFERRED
- Nice-to-have
- 興味深い調査対象

が残っていても Worker を走らせ続けない。

> **未処理 Work Item の存在は継続理由ではない。**

---

20. 成功指標

自己組織化の成功を「綺麗な分業」で測定しない。

見るもの:

- Time to First Human Feedback
- Time to First Evaluatable Build
- 総トークン消費
- Worker 起動数
- 重複調査量
- 同一ファイル再読込量
- 不要な Context 配布量
- 手戻り
- Reviewer 差し戻し
- Human Evaluation 前に不要化した実装・テスト量

追加指標:

```
Useful Work Ratio =
First Evaluatable Build へ直接寄与した作業
/
全 Worker 作業
```

厳密に数値化できなくても、改善方向として使う。

---

21. 最小ルール版

```
SELF-ORGANIZING WORKER RULES

- Worker は必要最小数だけ起動する。
- 新 Worker は Parallelism Gate を通る場合だけ追加する。
- 各 Worker は Compact Work Board を見て最重要の未担当作業を Claim する。
- 固定職種ではなく、その時点の Critical Path に応じて仕事を選ぶ。
- 理由のない重複を避ける。
- 独立検証時は他者の未確認仮説を渡さない。
- 同一領域への並行 Write を原則禁止する。
- 成果だけでなく失敗した試行も圧縮して共有する。
- Context は Need-to-Know だけ渡す。
- 担当変更は切替コストを上回る価値がある場合だけ。
- BLOCKED / UNRESOLVED を正常な結果として許可する。
- Worker は推論強度を勝手に上げない。
- First Evaluatable Build Gate を通過したら Worker Pool を停止する。
- Independent Reviewer には Worker 会話全文ではなく Review Packet を渡す。
```

Core Principle

> **Do not assign every role. Do not spawn every agent.**
>
> **Create the smallest environment in which useful roles can emerge.**

役割固定を減らすことと、Worker を増やすことは同義ではない。

Completion-First が「何を最適化するか」を決める。

Self-Organizing Worker Pool が「誰が何を行うか」を動的に決める。

Adaptive Reasoning が「その作業へどれだけ推論予算を使うか」を決める。

```
Completion-First
    → Goal / Scope / Stop / Phase
```

```
Self-Organizing Worker Pool
    → Dynamic work allocation / minimal parallelism

Adaptive Reasoning
    → Model / reasoning budget
```

この 3 層を分離し、

完成速度と総トークン効率の両方を改善すること

を v1.3 の目的とする。

---

22. Work Packaging との境界

Self-Organization は「誰が何を選ぶか」を決める。

Packetization は「その Worker へどの大きさで渡すか」を決める。

Worker へ自由を与えるために、能力不相応な Raw Work Item まで自由 Claim させる必要はない。

```
Parent/Sol:
Work Item 認識
→ Uncertainty 整理
→ Target Tier
→ Work Packet 作成

Worker Pool:
Eligible Packet を Observe
→ Claim
→ Work
→ Publish
→ Release
```

Packet が不適切だった場合は、Worker が無理に完遂しようとせず REPACKAGE_REQUEST を返す。

詳細は Adaptive-Work-Packaging.md を参照する。

Part V —Benchmark Telemetry

Benchmark Telemetry

Optional lightweight metrics for Completion-First comparisons

通常開発では有効化しなくてよい。v1 / v1.2 / v1.3 比較や Packet sizing 調整時のみ使う。

目的は推論ログを保存することではない。観測可能な作業量だけを記録する。

run_metrics.json

```
{
  "workflow": "completion-first-v1.3-redesigned",
  "result": "PASS",
  "elapsed_sec": 0,
  "first_evaluatable_build_sec": 0,

  "agents_spawned": 0,
```

```
  "packets_created": 0,
  "packets_one_shot_done": 0,
  "repackages": 0,
  "retries": 0,

  "files_read": 0,
  "files_changed": 0,
  "commands_run": 0,
  "tests_run": 0,
  "review_rounds": 0,

  "duplicate_work_detected": 0,
  "escalations": 0
}
```

events.jsonl

必要なら次のイベントだけ記録する。

```
AGENT_SPAWN
PACKET_CREATE
WORK_CLAIM
WORK_DONE
WORK_BLOCKED
REPACKAGE
TEST_RUN
TEST_FAIL
DEBUG_ATTEMPT
REVIEW_REJECT
ESCALATE
FIRST_EVALUATABLE_BUILD
```

長い自然言語ログ、Chain-of-Thought、全ファイル読込履歴は不要。

重要指標

```
One-shot Completion Rate
= one-shot DONE packets / completed packets

Repackage Rate
= repackages / packets

Retry Rate
= retries / packets

Cost per Successful Completion
= total actual cost / successful runs
```

Token/credit は Codex の自己申告ではなく、取得可能なら外部 usage データを結合する。