Meta-Skills：三层生命周期技能系统

架构设计、治理机制与部署模型

项目：claude-skills / meta-skills (public submodule)

Viska Wei

February 24, 2026

核心结论

Meta-Skills 是一个自包含、可扩展的 AI Agent 技能系统，采用 **3 层调用架构**（L0 入口 → L1 路径 → L2 原子能力）+ **8 阶段生命周期**（Discover → Knowledge）组织 **29 个原子能力**、**8 条路径模板**、**9 条质量策略**。系统通过严格的命名治理（18 受控动词 × 91 规范对象）和合约驱动（130 制品类型定义）实现可组合、可演进的技能编排。

关键指标：4 个 L0 命令 | 8 条 L1 路径 | 29 个 L2 能力 | 9 条策略 | 130 种制品类型 | 18 受控动词 | 91 规范对象

Contents

1 概述

1.1 项目定位

Meta-Skills 是 `claude-skills` 项目的**公开核心子模块**（Git submodule），提供领域无关的技能基础设施。它解决的核心问题是：

- **能力碎片化**——当 Agent 拥有数十甚至上百个原子操作时，如何让用户通过少量入口高效调用？
- **质量不可控**——如何在多步骤工作流中自动注入质量门禁，而不依赖人工检查？
- **命名混乱**——如何在技能持续增长时保持一致的命名、分类和可发现性？
- **系统退化**——如何让系统自身具备健康检查、熵清理、缺口诊断的能力？

Meta-Skills 通过**分层架构 + 治理约束 + 策略注入**三管齐下的方式系统性地解决上述问题。

1.2 设计原则

1. **分层可组合**（Layered Composability）——L0 薄路由、L1 编排模板、L2 原子能力，各层独立演进
2. **合约驱动**（Contract-Driven）——每个能力有 YAML 前置声明：输入/输出类型、前置条件、失败模式
3. **策略即代码**（Policy-as-Code）——质量检查提取为可复用的 YAML 规则，由解析器自动注入
4. **命名治理**（Named Governance）——受控动词表 + 规范对象表 + 强制命名模式
5. **自维护**（Self-Maintaining）——系统包含审计、改进和演化自身的工具
6. **隐藏设计**（Hidden-by-Design）——`_` 前缀目录对 Claude Code 自动发现不可见

2 系统架构

2.1 三层调用模型

Meta-Skills 的核心架构是一个**三层调用栈**，用户通过 L0 入口发出指令，系统逐层解析到具体的原子操作：

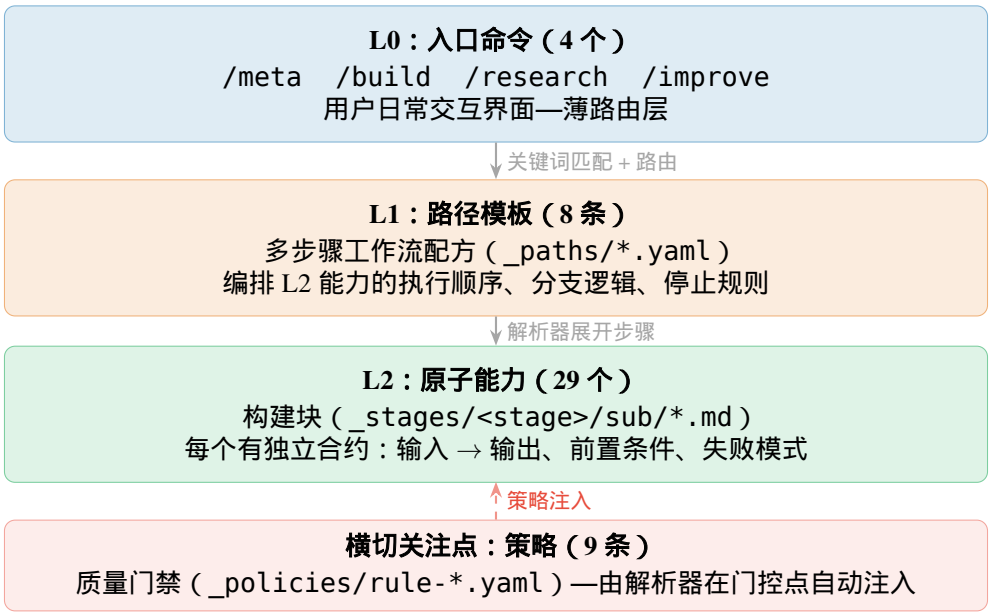

Figure 1: 三层调用架构示意图

2.2 八阶段生命周期

所有工作流都映射到一个统一的 8 阶段生命周期模型。每个阶段有明确的输入输出合约和门控条件：

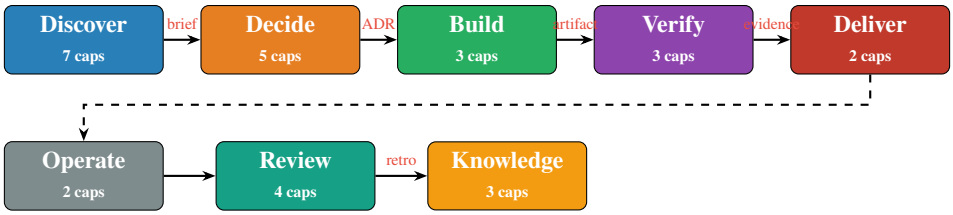

Figure 2: 8 阶段生命周期与门控制品

2.3 目录结构

系统采用**隐藏设计**——以 _ 前缀开头的目录不会被 Claude Code 自动发现，只有 4 个 L0 命令目录可被自动索引：

skills/ 目录布局

```
skills/
  meta/                    # L0: 系统自维护 (VISIBLE)
  build/                   # L0: 创建技能   (VISIBLE)
  research/                # L0: 科研流程   (VISIBLE)
  improve/                 # L0: 改进制品   (VISIBLE)
  _stages/                 # L2: 8 个生命周期阶段 (HIDDEN)
    discover/ decide/ build/ verify/
    deliver/ operate/ review/ knowledge/
  _paths/                  # L1: 路径模板 YAML (HIDDEN)
  _policies/               # 横切策略规则 (HIDDEN)
  _resolver/               # 解析器 + 治理文件 (HIDDEN)
  _tools/                  # 域工具族 (HIDDEN)
  _standards/              # 治理标准 (HIDDEN)
skills-registry.yaml       # 主注册表
```

2.4 解析流程

用户输入到最终执行的完整路径：

1. **L0 关键词匹配**——用户输入触发某个 L0 命令（如 `/research new`）
2. **L1 路径选择**——L0 内部路由表选择对应的路径模板（如 `path-research-new-experiment`）
3. **解析器展开**——Resolver 读取路径 YAML，逐步解析每个 `capabilities_needed`
4. **能力索引查找**——在 `capability-index.yaml` 中找到 `cap-*` 对应的 block 文件
5. **策略注入**——根据 step 的 `output_type`，从 `_policies/` 注入匹配的质量规则
6. **合约校验**——验证输入类型可用、前置条件满足
7. **执行 + 记录**——生成 `run-*` 执行记录

3 核心组件

3.1 L0 入口命令

系统提供 4 个用户可调用的 L0 命令，分为**系统命令**和**用户命令**两类：

Table 1: L0 入口命令一览

命令	类型	子命令数	路径数	职责
`/meta`	系统	4	4	技能系统自维护与演化
`/build`	用户	3 types	1	创建新的 L1/L2/规则
`/research`	用户	3	2	科研全生命周期管理
`/improve`	用户	2 modes	2	改进现有制品

3.1.1 /meta —系统自维护

`/meta` 是唯一作用于**技能架构本身**（而非用户制品）的命令，提供 4 个子命令：

- `health` → `path-general-skill-health`（12 步）——6 维度健康扫描：命名、合约、注册表、部署、覆盖率、去重
- `cleanup` → `path-general-entropy-cleanup`（11 步）——9 点一致性检查 + 自动修复
- `quality <skill>` → `path-general-skill-quality`（11 步）——审计目标技能是否符合 skill-creator-standard
- `gaps` → `path-general-capability-gap`（9 步）——能力缺口诊断 → 自动构建

安全保证

所有 `/meta` 操作遵循 PAUSE-before-action 模式——分析完成后必须等待用户确认才执行变更。评审循环最多 3 次迭代，无进展则停止。

3.1.2 /build —创建新技能

/build 负责在 3 层架构中创建新的构建块：

创建类型	命名模式	放置位置
L1 路径	path-<domain>-<outcome>.yaml	_paths/
L2 能力	cap-<verb>-<object>.md	_stages/<stage>/sub/
策略规则	rule-<scope>-<intent>.yaml	_policies/

工作流：解析意图 → [调研] → PAUSE → 脚手架 → 实现 → 验证 → 知识沉淀

3.1.3 /research —科研流程

/research 管理从研究问题到知识卡片的全生命周期：

- new → path-research-new-experiment（6 步）——结构化立项：RQ → 问题树 → 假设 → 指标 → 路线图 → 脚手架
- full → path-research-hypothesis-to-evidence（24 步）——完整循环：Discover → Decide → Build → Verify → Review → Knowledge
- card ——知识合成：回顾 → 原则提取 → 知识卡片 → Hub 同步

3.1.4 /improve —改进制品

/improve 对现有代码、文档、报告进行研究驱动的改进：

- loop（默认）→ path-general-improve-loop（9 步）——研究最佳实践 → 差距分析 → 改进 → 评审循环
- ratchet → path-general-ratchet-loop（5 步）——检查 → 修复 → 重新检查（最多 5 次迭代）

3.2 L1 路径模板

8 条路径模板是系统的**编排层**，每条定义了步骤序列、分支逻辑、停止规则和完成守卫：

Table 2: 8 条 L1 路径模板

路径 ID	步骤数	所属 L0	职责
path-general-skill-health	12	/meta	6 维度健康扫描
path-general-skill-quality	11	/meta	技能质量审计
path-general-entropy-cleanup	11	/meta	熵清理 + 一致性修复
path-general-capability-gap	9	/meta	能力缺口诊断与填补
path-general-improve-loop	9	/improve	研究驱动改进循环
path-general-ratchet-loop	5	/improve	轻量级棘轮修复
path-research-new-experiment	6	/research	实验立项规划
path-research-hypothesis-to-evidence	24	/research	假设到证据全循环

每条路径模板的 YAML 结构包含：

路径模板结构示例

```yaml
id: path-<domain>-<outcome>
steps:
  - stage: discover
    capabilities_needed: [cap-intake-brief, cap-extract-brief]
```

```
    output_type: brief
    gate_requires: [intake-manifest]
applicable_policies:
  required: [rule-quality-deliverable-minimum]
  recommended: [rule-completion-guard]
stop_rules:
  max_iterations: 3
  pass_criteria: "avg_score >= 4.0"
completion_guard:
  required_evidence: [evidence-bundle, improvement-log]
```

3.3 L2 原子能力

29 个原子能力分布在 8 个生命周期阶段中，每个都有独立的合约声明：

Table 3: L2 原子能力按阶段分布

阶段	数量	能力列表
Discover	7	cap-intake-brief, cap-extract-brief, cap-extract-requirements, cap-map-problem-tree, cap-map-hypothesis-tree, cap-extract-metrics-contract, cap-extract-standards-scout
Decide	5	cap-compare-option-matrix, cap-decide-adr, cap-plan-exec-plan, cap-plan-roadmap, cap-plan-experiment-design
Build	3	cap-map-tasks, cap-scaffold-experiment, cap-build-implementation
Verify	3	cap-plan-test, cap-assemble-evidence-bundle, cap-decide-quality-gate
Deliver	2	cap-package-output, cap-publish-release
Operate	2	cap-build-runbook, cap-build-observability
Review	4	cap-review-improvement, cap-decide-gate, cap-extract-retro, cap-extract-design-principles
Knowledge	3	cap-capture-card, cap-sync-hub, cap-sync-index
合计	**29**	

3.3.1 能力合约结构

每个 L2 能力以 Markdown 文件实现，头部包含 YAML 前置声明：

L2 能力合约示例 (cap-intake-brief)

```
---
cap_id: cap-intake-brief
verb: intake
object: brief
stage: discover
inputs:
  - goal-statement
  - url | file-path | repo-url
```

```
outputs:
  - intake-manifest
preconditions: []
side_effects:
  - "writes: artifacts/intake-manifest.md"
failure_modes:
  - "source_unreachable: URL or repo unavailable"
  - "ambiguous_goal: goal too vague to extract brief"
leveling: G3-V0-P2-M3
---
```

3.4 横切策略

9 条策略以声明式 YAML 定义，由解析器根据制品类型自动注入到门控点：

Table 4: 9 条质量策略

策略 ID	触发条件	检查项
rule-quality-deliverable-manifest	所有输出	证据包存在、非空、可复现
rule-completion-guard	横切	所有步骤执行、证据存在、PASS
rule-improve-verify-result	改进日志	标准包 (≥3 源)、选项矩阵、无回归
rule-skill-build-gate	技能制品	命名规范、合约有效、注册表
rule-skill-health-gate	健康仪表板	6 维度评分、阈值标准
rule-entropy-cleanup-gate	熵报告	确定性验证、幽灵引用检测
rule-capability-gap-detection	缺口分析	分类、影响 × 工作量评分
rule-research-front-loading	科研制品	假设驱动、指标合约、止损规则
rule-layer-dependency	所有路径	层间依赖合规（无跨层捷径）

4 治理机制

4.1 命名治理

系统通过**受控词汇表**强制执行一致的命名：

4.1.1 18 受控动词

所有 L2 能力的动词必须来自以下受控表：

1. `intake`
2. `extract`
3. `map`
4. `compare`
5. `decide`
6. `plan`
7. `scaffold`
8. `build`
9. `render`
10. `assemble`
11. `package`
12. `publish`
13. `check`
14. `track`
15. `triage`
16. `review`
17. `capture`
18. `sync`

每个动词定义了别名（用户同义词）和阶段边界（哪些阶段可以使用该动词）。

4.1.2 91 规范对象

对象表定义了所有合法的能力对象名，按领域分组（research、paper、repo、webui、docs 等），每个对象有 `used_in` 引用。

4.1.3 命名规则

层	模式	示例
L2 能力	`cap-<verb>-<object>`	`cap-intake-brief`
L1 路径	`path-<domain>-<outcome>`	`path-research-new-experiment`
策略	`rule-<scope>-<intent>`	`rule-quality-deliverable-minimum`
执行记录	`run-<path>-<ctx>-<date>-<seq>`	`run-path-research-...-20260224-01`

检查规则：kebab-case only | 3–6 token | 动词必须来自受控表 | 对象必须来自规范表 | 正确前缀

4.2 制品类型系统

`artifact-types.yaml` 定义了 130 种制品类型，涵盖：

- **原始输入**：`goal-statement`, `url`, `file-path`, `repo-url`
- **阶段输出**：`brief`, `intake-manifest`, `adr`, `evidence-bundle`, `knowledge-card`
- **横切制品**：`improvement-log`, `entropy-report`, `health-dashboard`

制品类型用于解析器的**类型兼容性验证**——步骤 n 的输出必须满足步骤 $n+1$ 的输入要求。

4.3 等级系统（Leveling）

每个能力标记一个 `Gx-Vy-Pz-Mk` 标签，用于解析器排序和系统健康评估：

维度	范围	含义
G (通用性)	G0–G3	G0= 临时 → G3= 核心跨域
V (易变性)	V0–V3	V0= 稳定 → V3= 快速变化
P (成熟度)	P0–P3	P0= 草稿 → P3= 硬化
M (观测度)	M0–M4	M0= 存根 → M4= 生产验证

核心能力的典型等级为 `G3-V0-P2-M3`（核心、稳定、生产就绪、已观测使用）。

5 解析器机制

5.1 解析算法

解析器（`_resolver/resolver.md`）执行 7 步解析流程：

1. **加载路径模板**——读取 `_paths/<path_id>.yaml`，验证结构完整性
2. **逐步解析**——对每个步骤：
   a. 在 `capability-index.yaml` 中查找 `cap_id`
   b. 验证动词来自 `verbs.yaml`，对象来自 `objects.yaml`
   c. 如果精确匹配失败，启动**模糊搜索**
   d. 多提供者时按 P/M 等级排序
3. **模糊搜索**（降级查找）——按动词 + 对象模式 → 按输出类型 → 按输入类型 → 按阶段
4. **加载策略**——`required` 策略必须注入，`recommended` 按上下文注入

5. **插入策略检查**——匹配策略的 `triggers.output_types` 与步骤的 `output_type`
6. **生成执行 ID**——格式：`run-<path>-<ctx>-<yyyymmdd>-<seq>`
7. **返回解析后的执行计划**

5.2 治理文件

解析器依赖 5 个治理文件：

文件	条目数	职责
`verbs.yaml`	18	受控动词 + 别名 + 阶段边界
`objects.yaml`	91	规范对象 + 领域标签 + 引用
`artifact-types.yaml`	130	制品类型定义（输入/输出）
`capability-index.yaml`	29	cap-* → block 文件映射
`resolver.md`	—	解析算法规范

6 关键设计模式

6.1 门控检查工作流

每条路径在关键动作前设置显式门控：

- **PAUSE 点**——分析完成后等待用户确认方向
- **Verify 步骤**——必须 PASS 才能继续
- **Critic 循环**——迭代直到 PASS 或预算耗尽
- **Completion Guard**——强制所有步骤完成

完成守卫机制

`rule-completion-guard` 通过状态文件 `.claude/completion-guard.local.md` 追踪执行进度。当且仅当所有步骤完成、所有证据存在、验证通过时，输出 `<promise>ALL_STEPS_COMPLETE</promise>` 信号。

6.2 多分支路径

路径支持条件分支：

- `decision-gate` → “continue”（循环）/ “change direction”（转向）/ “publish”（外部发布）
- `quality-gate` → “PASS” / “FAIL”（重试）
- `critic-review` → “ITERATE”（循环）/ “PASS”（完成）/ “BUDGET_EXHAUSTED”（停止）

6.3 组合优于继承

系统采用扁平组合模式而非层次继承：

- 每个 L2 能力是独立的——可以单独运行
- L1 路径通过编排组合 L2 能力
- 策略环绕在步骤周围（不嵌入能力内部）
- L0 命令是薄路由器（逻辑在路径中，不在命令中）

6.4 扩展点设计

在 path-research-hypothesis-to-evidence 中预留了领域扩展点：

扩展能力 ID	用途
cap-build-data-pipeline	领域特定数据处理
cap-build-model-loss	模型架构定义
cap-render-eval-harness	评估脚本生成
cap-track-experiment	实验自动记录
cap-check-metric-sanity	指标鲁棒性检查
cap-check-reproducibility	可复现性验证
cap-plan-resource-budget	计算资源预算

这些扩展点由私有仓库实现，核心模块不包含它们的具体实现。

7 部署模型

7.1 部署脚本

tools/setup.sh 将 meta-skills 部署到 ~/.claude/skills/，供 Claude Code 使用：

1. 复制 4 个 L0 命令目录（使用 cp -rL 解引用符号链接）
2. 复制 8 个生命周期阶段（_stages/）
3. 复制架构层（_paths/, _policies/, _resolver/, _standards/）
4. 创建符号链接（_tools/, skills-registry.yaml）
5. 验证部署完整性（检查文件存在、计数）

关键限制

Claude Code **无法**通过符号链接发现 SKILL.md 文件。解决方案：对技能目录使用 cp -rL（解引用复制），仅 _tools/ 和 skills-registry.yaml 保留为符号链接。

7.2 子模块集成

Meta-Skills 设计为 Git submodule 使用：

子模块集成方式

```
# 添加为子模块
git submodule add <url> meta-skills

# 私有仓库的 setup.sh 应：
# 1. 先部署核心（bash meta-skills/tools/setup.sh）
# 2. 覆盖领域扩展（复制额外的 L0/L1/L2）
# 3. 使用扩展的解析器（verbs/objects 是核心的超集）
```

7.3 验证工具

系统提供 4 个验证脚本：

工具	行数	职责
`setup.sh`	120	部署到 ~/.claude/skills/
`build_capability_index.sh`	92	从 block 文件重建 capability-index
`validate_contracts.sh`	172	验证 YAML 前置合约
`validate_aliases.sh`	149	验证触发词 + 能力引用

8 定量分析

8.1 系统规模统计

Table 5: Meta-Skills 系统规模

类别	说明	数量
L0 入口命令	用户直接调用	4
L1 路径模板	多步骤编排	8
L2 原子能力	构建块	29
横切策略	质量门禁	9
受控动词	命名治理	18
规范对象	命名治理	91
制品类型	合约系统	130
部署脚本	工具链	4
治理密度	策略 / 能力	0.31
编排比	路径步骤 / 能力	2.90
词汇覆盖	动词 × 对象	1,638

8.2 阶段覆盖分析

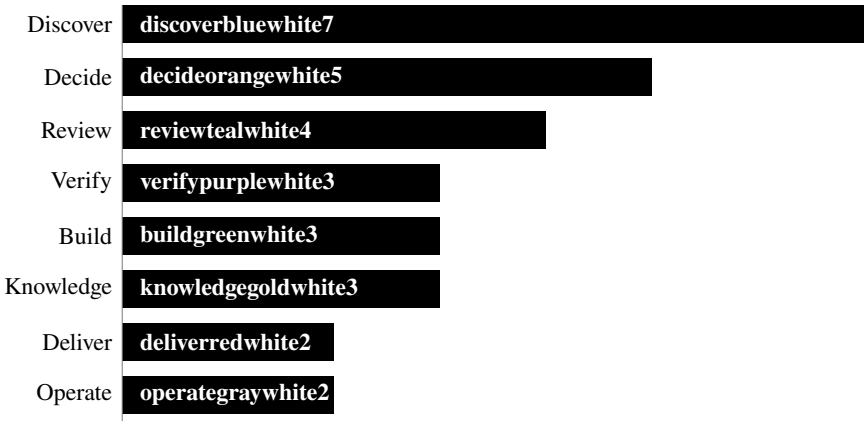

Figure 3: 各阶段能力覆盖数（横条图）

观察：

- **Discover**（7 个）覆盖最密——反映" 前端加载" 设计哲学（在动手前充分理解问题）
- **Decide**（5 个）次之——决策阶段需要多种比较和规划工具
- **Deliver/Operate**（各 2 个）最少——这些阶段更多依赖外部工具和域特定实现
- **Review**（4 个）包含评审、门控、回顾、原则提取——知识闭环的关键

8.3 等级分布

等级	数量	占比
G3-V0-P2-M3（核心稳定已验证）	18	62.1%
G3-V1-P2-M3（核心、微变、已验证）	5	17.2%
G3-V1-P1-M2（核心、微变、试点）	2	6.9%
G2-V1-P1-M1（多域、新建）	2	6.9%
G3-V0-P1-M2（核心、稳定、试点）	1	3.4%
G3-V0-P3-M3（核心、稳定、硬化）	1	3.4%

唯一达到 P3（硬化）等级的是 `cap-decide-quality-gate`——作为质量最终裁决者，它本身需要最高的可靠性。

9 与私有仓库的关系

Meta-Skills 作为公开子模块，与私有仓库形成**核心-扩展**架构：

Table 6: 公开核心 vs 私有扩展

类别	公开核心	私有扩展
L0 命令	4	+8（含 workflow, office 等）
L1 路径	8	+19
L2 能力	29	+40
策略	9	+15
插件	0	11
合计	**50**	**+93**

私有仓库通过以下方式扩展核心：

1. 添加领域特定的 L0 命令（如 `/read`, `/write`, `/search`）
2. 实现核心预留的扩展点能力（如 `cap-build-data-pipeline`）
3. 添加领域策略（如 `rule-webui-dev-server-verify`）
4. 添加插件技能（如 docx, pdf, pptx, xlsx 等办公工具）
5. 扩展受控词汇表（更多动词和对象）

10 总结

10.1 架构优势

Meta-Skills 通过 3 层调用架构实现了以下目标：

1. **简洁的用户界面**——4 个 L0 命令覆盖了系统维护（meta）、创建（build）、科研（research）、改进（improve）四大类需求
2. **可靠的质量保证**——9 条策略自动注入到门控点，无需人工记忆检查清单
3. **严格的命名一致性**——18 动词 $\times$ 91 对象的受控词汇表消除了命名歧义
4. **灵活的扩展性**——公开核心 + 私有扩展的子模块模式，领域能力可独立演进

5. **自维护能力**——`/meta` 命令让系统能诊断和修复自身的健康问题
6. **可追溯性**——合约驱动的类型系统和执行记录确保每个决策可追溯

10.2 演进方向

基于当前状态，系统的潜在演进方向包括：

- `rule-harness-safety.yaml`——meta 操作安全护栏（已列入待办）
- 独立部署测试——验证公开仓库的 `setup.sh` 在全新环境下的可用性
- 设计手册更新——反映从 1 L0 到 4 L0 的架构升级
- 更多 P3（硬化）能力——当前仅 1/29 达到 P3 等级

一句话总结

Meta-Skills 是一个**合约驱动、策略注入、自维护**的 AI Agent 技能基础设施，通过 3 层分离（入口 / 编排 / 原子）+ 8 阶段生命周期 + 严格命名治理，实现了复杂 Agent 能力的可组合管理。