ENTERPRISE EXECUTION ARCHITECTURE

• 18 页深度图解

WORKFLOW SYSTEMS / 01

工作流与 Agent 架构

Dynamic Workflow × **Ontology**

Agent Loop · 多 Agent 协作

企业 AI 执行系统全景

BUSINESS → ACTION → FEEDBACK

业务系统与事件 SYSTEM OF RECORD

01 **Ontology + Policy**
业务语义与规则

02 **Context + Compiler**
上下文与流程生成

03 **Harness + Runtime**
运行、恢复与审计

04 **Agent + Action**
判断与受控执行

05 **Eval + Feedback**
评测与持续改进

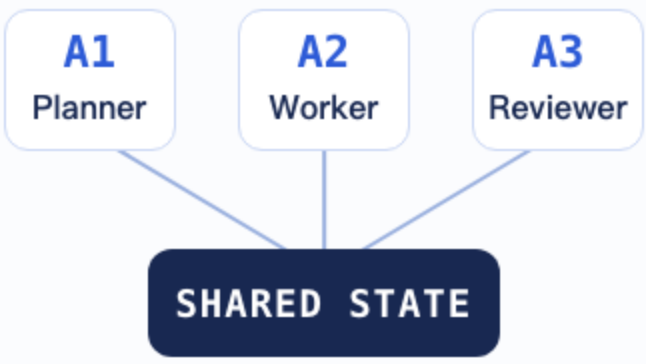
Agent Loop

单 Agent 的闭环执行



多 Agent 协作

角色分工 · 共享状态 · 有序交接



并行 · 串行 · 交接

心晴九天 | 企业 AI 学习资料

扫码体验我的 Agent，或关注小红书获取后续拆解

Dynamic Workflow x Ontology / Public Study Pack

我的 Agent

小红书

先给结论

企业 AI 的关键，不是让 Agent 更自由

而是把一次业务任务，拆成三种不同的责任。

CODE

稳定规则交给代码

循环、分支、校验、重试、预算和停止条件。它负责让流程可以复现。

AGENT

含糊判断交给 Agent

理解语义、比较证据、处理冲突和权衡方案。它只处理真正的不确定性。

POLICY + HUMAN

高风险动作交给权限和人

合同、付款、授权、外发和不可逆副作用。前置条件不满足，就不能执行。

真正有价值的，不是一群 Agent 自己聊着把事做了，而是一条由 Harness / Runtime 承载、能检查、能暂停、能恢复、能评测的执行链。

先看一个合同审核事故

Agent 审到一半中断，恢复后会发生什么？

它已经标记了三条风险、发出一次审批，但结果还没完整返回。系统重启以后，如果计划、状态和副作用只存在对话里，就很难知道哪些步骤应该复用、哪些必须重跑。

- × 已完成判断和未完成判断混在上下文里
- × 恢复后可能重复发审批、写数据库或通知人
- × 新旧政策冲突，却没有版本化依据
- × 无法证明：谁基于什么证据做了哪一步

这不是模型不聪明。真正的问题是：计划、运行状态和真实世界副作用都没有成为系统的一等公民。

合成事故案例 | 用于解释工程风险

真正的区别：谁持有计划

Skill、Agent、Agent Team、Workflow 不是一回事

不要只看名字里有没有“多 Agent”。关键是：下一步由谁决定，中间结果存在哪里，编排本身能不能复用。

SKILL

模型遵循说明

计划主要留在上下文；可复用的是说明，不是整条执行路径。

AGENT / SUBAGENT

模型逐轮决定

根据工具结果动态前进，适合开放判断与局部探索。

AGENT TEAM

Lead Agent 持有计划

同伴并行工作，由主 Agent 分工、协调并综合结果。

WORKFLOW

脚本持有计划

循环、分支和中间变量进入代码，编排本身可读、可重跑。

企业通常不是四选一：Workflow 控制大结构，Agent 处理不确定性，Skill 提供方法，Team 承担高价值并行探索。

官方机制 | Claude Code Dynamic Workflows

DYNAMIC WORKFLOW

模型为当前任务写脚本，Runtime 再执行它

下面使用的是当前 Claude Code Workflow 支持的核心形式：脚本调用 `agent()`、`pipeline()` 或 `parallel()`，中间结果留在变量里。

```
const found = await agent(
  "列出合同中的高风险条款", { schema }
)

const reviews = await pipeline(
  found.clauseIds,
  id => agent(`审核条款 ${id}`)
)

return reviews.filter(Boolean)
```

> **当前产品边界：**Claude Code Workflow 运行中不能等待普通用户输入。需要人工签字时，应在下一阶段或外部治理系统中完成。

来源观点 | 李昕公开文章　官方机制 | Claude Code

代码与 AGENT 的责任边界

让程序负责秩序，让模型负责理解

这不是“规则任务用代码、复杂任务用模型”这么简单，而是把一个任务里的确定性和不确定性拆开。

CODE / RUNTIME

确定性控制

- 循环、分支、并发
- 字段校验与停止条件
- 预算、限流、Checkpoint
- 幂等键与失败策略

AGENT / MODEL

语义性判断

- 理解条款真实含义
- 比较互相冲突的证据
- 识别模糊需求的目标
- 在多个方案间权衡

再进一步：高能力模型负责生成或修改 Workflow；不同节点再使用“足够好且成本合适”的模型。前提是每个节点都有可重复的 Eval。

来源观点 | 李昕公开文章　当前机制 | Claude Code

最容易被忽略的工程边界

能恢复结果，不等于能回滚副作用

Claude Code 会记录运行中的 Agent 结果并支持同一会话内重放，但恢复不是虚拟机快照，也不是数据库事务。

COMPLETED

复用已保存结果

输入未变化的已完成 Agent，可以直接返回之前的结果。

RUNNING / FAILED

重新执行

中断或失败的 Agent 会重跑；之后启动的节点也可能再次执行。

NEW SESSION

通常从头开始

新会话没有旧运行可恢复；必须把状态留在可访问的持久层。

Agent 已经发出的消息、追加的文件、写入的数据库记录，不会因为 Workflow 重放而自动撤销。

用“目标状态”描述任务，避免重复追加

写操作携带业务幂等键或请求 ID

隔离工作区，写后重新读取并验证

原文重点 | Journaled Replay

官方机制 | Claude Code

AGENT 不是一次 PROMPT

真正的 Agent，是一个带工具的 Loop



Session　Sandbox　Storage　Context　Trace

`agent()` 不是问模型一次。定义本身是“模型使用工具的 Loop”；Session、Sandbox、存储、Context 管理和 Trace，则是让它长期可靠运行的 Harness 能力。

多个 AGENT 怎么协作

Parallel 和 Pipeline，千万别混用

两者解决的不是同一个问题：Parallel 需要一个“等待屏障”，保证多份判断到齐再聚合；Pipeline 追求连续吞吐，一项完成就立刻流向下一步。

PARALLEL

全部到齐，再继续



适合多视角审查、投票和综合判断。

PIPELINE

一项完成，立刻前进



适合批量迁移、逐项加工和流式处理。

并行不是越多越好。错误的并发会放大成本、冲突与噪音。

AGENT 之间的接口

不要靠自然语言接力，要靠结构化契约

自然语言适合人读，却不适合作为系统接口。上游写得再像报告，下游也可能误读字段、证据位置或缺失信息。

```
{
  "clause_id": "C-042",
  "risk_level": "high",
  "finding": "责任范围不明确",
  "evidence_refs": ["p12-l08"],
  "confidence": 0.86,
  "missing_info": []
}
```

Schema 保证什么?

字段、类型、缺失标记和下游可读性。

Schema 不保证什么?

判断是否正确、证据是否充分、结论是否过期。

格式稳定只是起点。语义正确还需要证据、规则与 Eval。

只有 WORKFLOW 还不够

流程会跑，不等于它懂业务世界

CRM、合同库、财务和审批系统仍然是事实与交易的来源。Agent 真正缺的，是跨系统理解“当前对象是谁、处于什么状态、哪条政策生效、谁有权改变它”。

> **Ontology 不是替代现有系统，**而是在 System of Record 之上提供稳定的业务语义、关系、状态和受治理动作。它是经常缺失的关键业务层之一。

CRM　合同库　审批　财务　知识库　MCP Tools

关键业务语义层

对象 · 关系 · 状态 · 规则 · 权限 · 证据

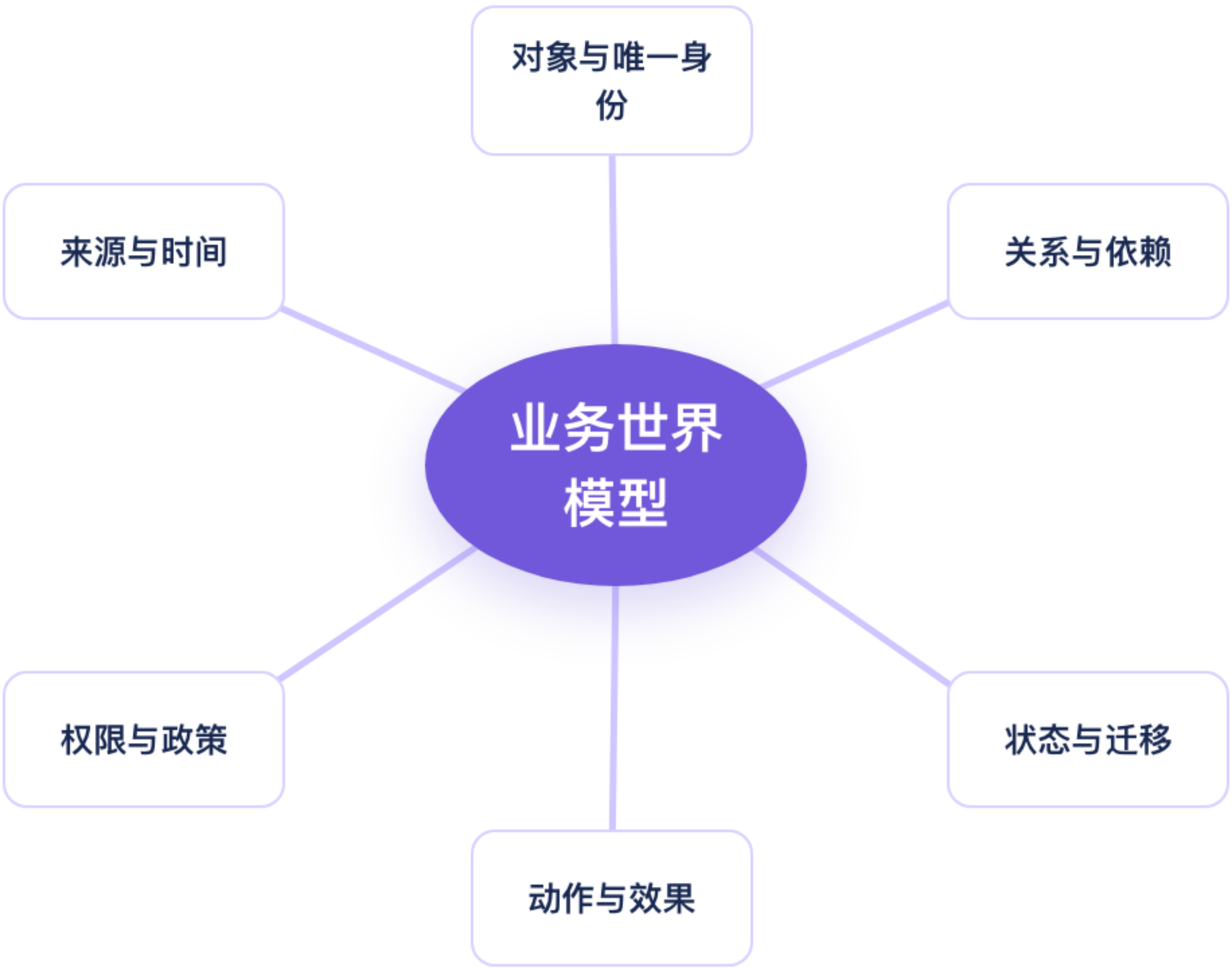
“请审核这份合同，并决定下一步”

本文推演 | 不是李昕原文直述

ONTOLOGY 真正补上的东西

Ontology 不是一个“高级知识库”

它是业务世界的稳定模型，也是 Workflow Compiler 使用的**业务类型系统与动作语法。**知识库主要回答“有什么材料”，Ontology 还要回答“这是什么对象、现在能做什么、由谁来做”。



架构推演 | 结合 Palantir Ontology 官方机制

CONTEXT ENGINEERING × ONTOLOGY

不是把整个知识库塞进 Prompt

前沿做法是 Just-in-time Context：只提取与本次目标有关、带版本的业务快照。这样每次生成都能说明：依据了哪批对象、哪版政策和哪一刻的状态。

> **GoalSpec + EntitySnapshot + PolicySet + ToolCatalog + RiskBudget → Workflow Compiler**

EntitySnapshot

- 相关对象
- 对象关系
- 当前状态
- 适用政策
- 可用动作
- 权限与证据时效

→

Compiler 输出

- Workflow IR / Code
- Input / State Schema
- Agent Contract
- Side Effect Plan
- Checkpoint Plan
- Human Approval Plan

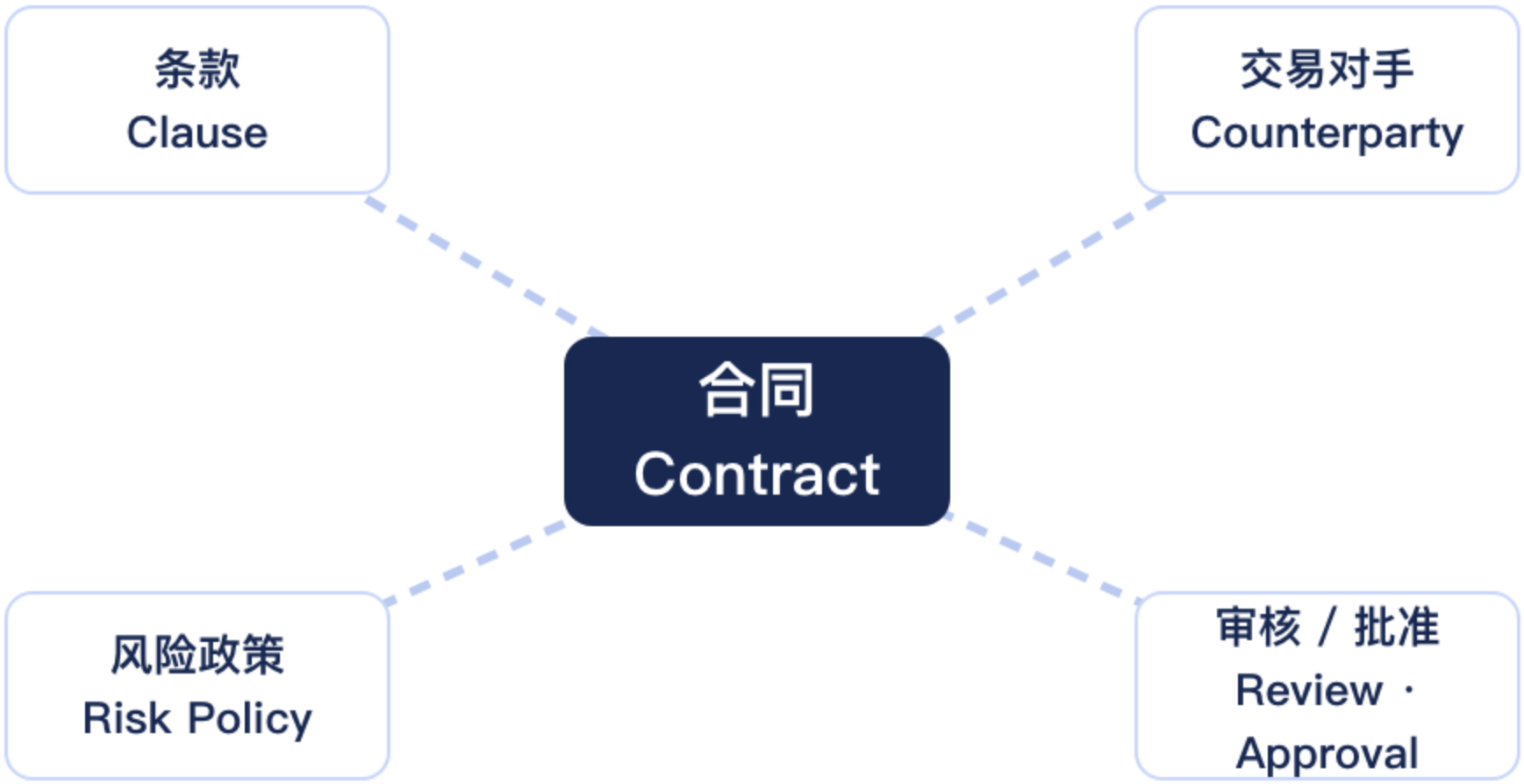
官方方向 | Anthropic Context Engineering

本文推演 | Workflow Compiler 输出契约

用合同审核跑一遍

先让系统认识这个“合同世界”

在自动审核之前，先定义合同、条款、交易对手、风险政策、审核记录和批准记录之间的关系，并明确每个动作的前置条件与执行权限。



草稿 → 审核中 → 待修改 → 已批准

标记风险

有可定位证据

请求修改

高风险未解决

批准合同

权限与审核齐备

合成演示案例 | 非真实客户成果

再为这一份合同生成流程

Ontology 稳定，Workflow 随任务变化

底层业务语义保持一致，具体执行路径却可以按合同类型、风险等级、缺失材料和政策版本动态变化。

- CODE 检查版本、必填字段、日期和证据时效
- AGENT 识别含糊条款，输出结构化 ClauseAssessment
- PARALLEL 多个 Agent 并行审核不同条款组
- CODE 合并、去重，检查每个判断是否有证据
- AGENT 处理冲突结论，并生成带证据的 Action Proposal
- HUMAN 权限与政策守门；法务批准后执行并写入日志

下一份合同可以走不同路径，但仍受同一套业务规则约束。

合成演示流程 | 非 Claude Code 内置审批能力

生成 ≠ 执行

计划生成权，不等于真实世界执行权

模型可以提出下一步，但不能因此自动获得改合同、付钱、发消息或改权限的能力。计划必须先进入确定性的治理层。

1	Schema 校验	VALIDATE
2	权限与状态机检查	AUTHORIZE
3	预算、快照与 Checkpoint	CONTROL
4	副作用与业务幂等键	PROTECT
5	高风险动作人工确认	APPROVE

```
ActionLog { // 示意字段 actor: "legal-reviewer-07",
workflow_version: "wf_2026_09_28_03", action: "ApproveContract",
object: "contract_1042", occurred_at: "2026-09-28T16:20+08:00" }
```

高风险写操作调用受治理的 Action，而不是让 Agent 直接写数据库。Palantir Action Log 记录成功提交；失败、耗时与成功率由 Metrics / Trace 追踪。

官方机制 | Palantir Actions / Action Log

它能重构所有业务吗

不能。至少要满足 6 个条件

这套架构最适合“过程复杂但边界可描述、价值足够高”的业务。技术上能做，不等于经济上值得做。

01

业务对象与状态可以被识别

02

关键动作能通过工具或 API 执行

03

结果能用规则、测试、证据或 Rubric 验证

04

权限与外部副作用能够显式建模

05

任务频率、价值与成本足以支撑投入

06

失败可以恢复，并且始终有人负责

如果目标不清楚、数据不可用、结果无法验证，或者失败根本无法补偿，这套架构只会让系统更快、更规模化地犯错。

可被建模、调用、验证和恢复的业务，才真正有机会被重构。

最后一张架构图

企业 Agent 真正缺的，是一个完整执行闭环

它不只生成计划，还必须管理上下文、受控运行、恢复失败、评测结果，并把反馈重新写回业务模型与政策。

业务系统与事件	文档、数据库、会话、实时事件与政策版本
Ontology + Policy	对象、关系、状态、合法动作、权限与证据来源
Context + Compiler	按目标提取快照，并生成受限的 Workflow IR / Code
Harness + Runtime	Sandbox、调度、Checkpoint、恢复、预算与 Trace
Agent + Action	Agent 处理必要判断，受治理 Action 改变业务状态
Eval + Feedback	验证结果与副作用，把错误和新事实反馈给 Policy / Ontology

真正的壁垒不是接入更多模型，
而是把业务语义、执行约束和反馈，变成持续进化的系统。

你所在的公司，哪条流程同时满足这 6 个条件?

如何使用这份资料

先看架构边界，再看运行机制，最后用检查表判断自己的业务是否适合。

01

先建立边界

第 2-5 页

先判断由谁持有计划：Skill、Agent、团队或 Workflow。

02

理解运行机制

第 6-10 页

看代码与模型如何分工，以及 Replay、Parallel、Pipeline、Schema。

03

补齐业务语义

第 11-15 页

理解 Ontology、Context Compiler 与合同案例如何连接。

04

做落地判断

第 16-18 页

检查权限、验证、恢复、评测和投入产出，不为架构而架构。

学习时只追问一个问题：

这一步应该由确定性代码负责，还是需要 Agent 处理语义与权衡？如果失败，谁能发现、恢复并承担责任？

核心概念速查

这些词容易被混用。下面给出本资料采用的工作定义，不把厂商实现混成同一个概念。

Workflow	由代码或显式图结构持有主要计划，适合稳定步骤、分支、循环、校验和预算。
Dynamic Workflow	围绕当前任务动态生成或调整工作流代码，再交给 Runtime 验证与执行。
Agent Loop	模型在观察、推理、行动、反馈之间循环，适合处理语义理解和不确定判断。
Multi-Agent	多个隔离角色通过共享状态、并行、串行或交接协作，不等于自由群聊。
Ontology	对业务对象、关系、状态、规则、权限和动作的统一语义建模。
Harness / Runtime	承载会话、工具、上下文、并发、恢复、审计、预算与执行边界。
Eval / Feedback	用规则、测试、Rubric、指标和人工反馈判断结果，并反向改进系统。

落地前检查清单

技术上能实现，不等于经济上值得。六项条件同时成立，才值得进入工作流重构。

☐ 业务对象可识别

对象、关系和关键状态能被稳定描述。

☐ 关键动作可调用

系统提供
API、工具或可治理的操作入口。

☐ 结果可以验证

有规则、测试、证据或 Rubric
判断质量。

☐ 权限能够建模

高风险动作有前置条件、审批和最小权限。

☐ 投入产出成立

任务频率、价值和成本足以支撑建设。

☐ 失败可以恢复

具备幂等、状态检查、重试、补偿和责任人。

停止条件

目标不清、数据不可用、结果无法验证，或者失败无法补偿时，先修业务基础，不要急着增加 Agent。

架构练习画布

选一个真实业务任务，按下面六格写出最小可执行版本。空白处可以直接打印填写。

01 业务对象与状态

核心对象是谁？有哪些关键状态与关系？

02 代码持有的流程

哪些步骤必须稳定、可复现、同输入同输出？

03 Agent 判断

哪些步骤需要读懂语义、比较证据或处理歧义？

04 动作与权限

哪些动作影响资金、权限、外部系统或客户？

05 验证与恢复

如何校验结果？失败如何重试、补偿或人工接管？

06 Eval 与反馈

用什么指标、样本和人工反馈持续改进？

一手来源与分享边界

本资料是独立学习笔记，不代表李昕、飞书或任何厂商立场。
外部事实请以原始页面为准。

原文 李昕：Harness 101 - 复刻 Dynamic Workflow（含代码）
my.feishu.cn/wiki/TM25wR9ozih8yRkaAVKcFjP1nMq

Claude Claude Code Workflows
code.claude.com/docs/en/workflows

Anthropic Effective context engineering for AI agents
www.anthropic.com/engineering/effective-context-engineering-for-ai-agents

Anthropic Code execution with MCP
www.anthropic.com/engineering/code-execution-with-mcp

OpenAI Harness engineering
openai.com/index/harness-engineering/

Palantir Foundry Action log
www.palantir.com/docs/foundry/action-types/action-log

转发建议

可以分享原文链接与本学习资料。请保留作者和来源，不复制原文全文，不把外部代码、截图或观点冒充原创；商业使用前另行取得授权。