Skills 真香，但非Claude模型怎么办？

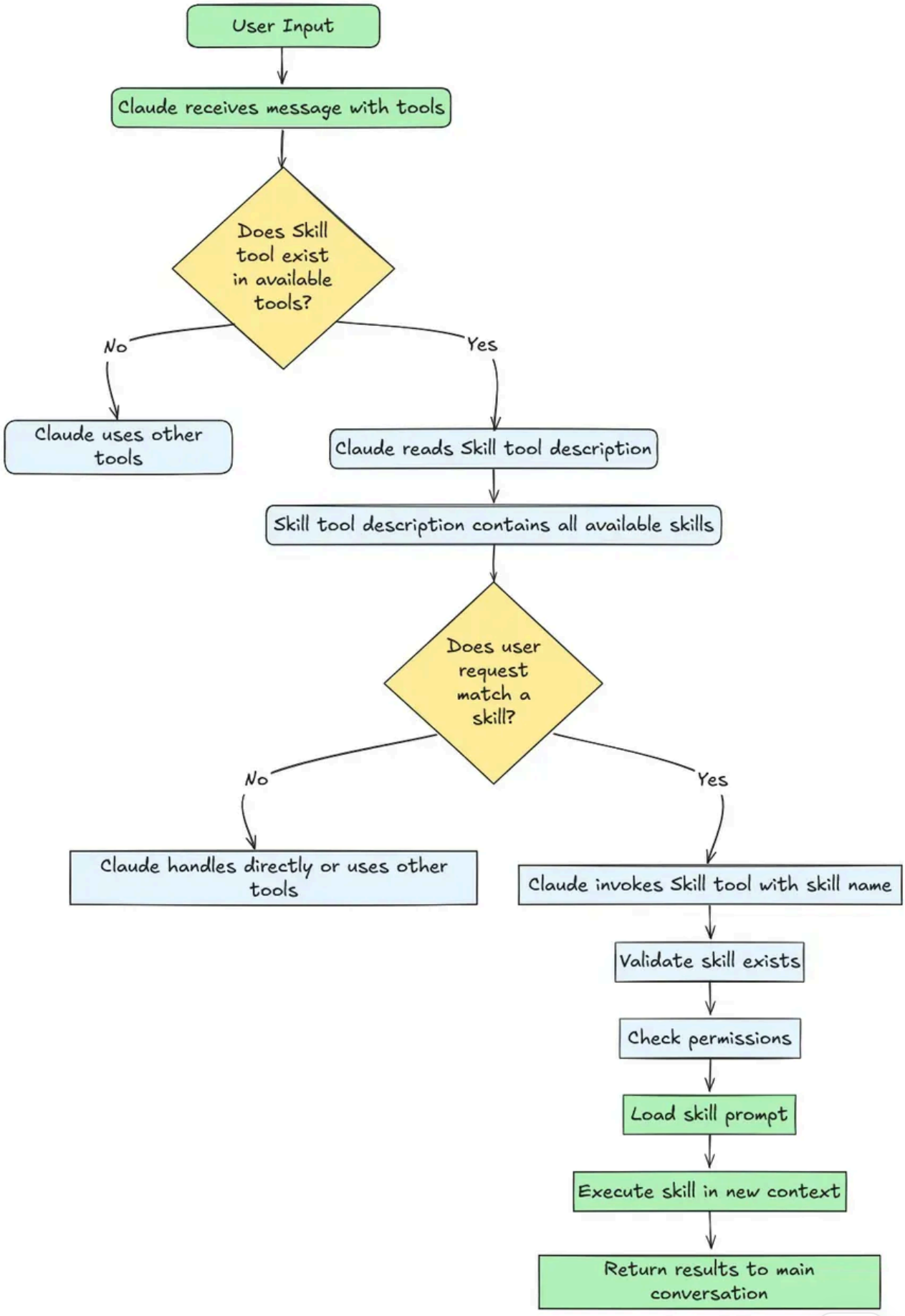
原创 叶小钗 叶小钗 2026年1月6日 08:28 四川

> AI训练营**7期**，**1月下旬**开班，欢迎咨询

关于什么是Skills，我们在之前的文章已经有过详细介绍，他是Anthropic在Claude模型中引入的一种新机制，可以理解为一组封装好的技能包。

每个技能包包含：说明文档、脚本和资源。Claude 在需要时会动态加载相应技能，以提高模型在特定任务上的一致性和表现。

与传统一次性提示不同，Skills 采用按需的逐步迭代加载机制：当用户提出请求时，Claude 会扫描所有可用技能，根据语义匹配自动选择相关技能，并仅加载完成任务所需的指令和资源。整体流程如图所示：



总的来说，Skills 从工程实践出发，显著改善了 Agent 在调用工具方面面临的诸多痛

点：

一、工具调用准确性提升

传统 Agent **如果不做工程层面的主动优化**，往往一次性注册大量工具接口，这会导致模型容易混淆相似的工具名称或参数，从而出现调用错误。

Skills 引入语义过滤和专用技能包，让模型每次仅关注当前任务相关的少数工具或指令，从源头上减少歧义，结果是整个工具调用准确率上来了。

只不过，技能如果多了后，其中一定会碰到类似的skill，之前Tools意图识别的问题，也有可能再发生。

二、流程一致性

每个 Skill 都以明确的SOP定义任务步骤和内部逻辑，使模型按照既定流程执行，减少自由发挥带来的偏差，这相当于将 **Workflow** 从代码挪到了提示词，或者**Skill**。

instructions通常包含多轮交互的步骤拆解、错误检查和决策条件，这种结构化流程让复杂任务的执行更加可靠；

同时，Skills 还能通过上下文隔离来提高稳健性：不同skill各司其职，互不干扰，某一步的错误不易“传染”到整个对话流程.

三、提示词的可维护性

在之前，提示词膨胀其实是个很烦的工程问题，有了 Skills，开发者无需将所有可能的指令和规则预先写入一个庞大的系统提示中。

取而代之的是，将知识和流程按场景拆分到独立skill中，只有在相关任务时才动态加载。这种模块化按需加载既节省了上下文，又使提示词的维护更简单，新增或修改某项流程只需更新对应的skill文件即可，而不用到处去烦躁代码中的提示词。

这里与第二点一样，如果skill过多，依旧会有新的工程维护问题。

综上：Skills 通过 封装**SOP+**按需加载 的工程方式，使 AI Agent 的工具使用更加精准、维护成本更低了。

只不过，前面也说了如果skill数量过多，依然会产生新的维护工程问题；另一方面当前很多模型还不支持**Skills**，所以我们今天简单用代码模拟实现，这样可以让大家更清楚两点：

1. 第一，Skills真的是一种针对Agent的工程优化；
2. 第二，为什么skill数量过多依旧会产生工程维护问题；

OpenAI如何使用Skills

Claude Skills 给人的体验提升，来自三件工程化的事：

1. **skill资产化：**SOP/规则/话术不再散落在 Prompt 里，而是变成可版本化的文件资产；
2. **先选再加载：**先路由到相关 Skill，再把这个 Skill 的指令加载进上下文；
3. **在 Skill 内约束工具使用：**让模型按 SOP 使用工具，减少乱调用、漏调用、乱序；

如果要自己实现简单的Skills，至少需要考虑上述场景。

一、Skills 怎么定义

整体结构设计完全照抄：

```
skills/
  expense_reimbursement/
    skill.json
    instructions.md
    examples.md         # 可选
    validators.py       # 可选：结果校验/兜底
```

skill.json，用于路由与治理：

```
{
  "name": "expense_reimbursement",
  "version": "1.2.0",
  "description": "查询员工差旅/报销数据，支持按项目拆分，输出财务口径总结",
  "tools": ["get_employee_info", "query_reimbursements", "analyze_reimbursement_data"],
```

```
  "owner": "finance-ai-team",
  "risk_level": "internal"
}
```

instructions.md。**核心：SOP + 工具使用说明**

```
触发条件：当用户提到“报销/差旅/项目拆分/财务口径”
步骤：
解析员工与时间范围（必要时追问补槽）
调 get_employee_info
调 query_reimbursements
若用户要求拆分，则调 analyze_reimbursement_data(group_by="project")
输出格式：一句话结论 + 关键数字 + 风险提示
```

至此，已经把“Workflow 从代码搬到了 Skill 里了”。基本配置整理结束，就需要准备全局架构了：

二、全局架构

全局架构的话，依旧是模拟Claude做实现，可以分拆为四个模块：

1. **Skill Registry：**加载 skills/ 目录，得到所有 skill 元数据与 SOP 文档；
2. **Skill Router：**根据用户问题，从技能库里选最合适的skill
3. **Skill Loader：**把选中的 SOP 注入系统；
4. **Agent Executor：**用 OpenAI Function Calling 调工具，按 SOP 走完整流程；

这里的关键点是**Router 与 Executor 分离**

1. **Router：**用便宜模型/小调用做“分类/路由”
2. **Executor：**用主模型做“推理 + 工具调用 + 生成”

这是工程上把延迟与成本压下来的常用做法，很多真实生产系统都是这样在玩。我们这里没可能全部做实现，就给大家展示下最小伪代码即可：

三、最小伪实现

首先是Skill Registry模块代码大概长这样：

```
def load_skill_registry():
  skills = []
  for folder in list_folders("skills/"):
    meta = read_json(f"{folder}/skill.json")
    sop  = read_text(f"{folder}/instructions.md")
    skills.append({**meta, "sop": sop})
  return skills
```

再然后就是最为关键的“意图识别”，选择加载了，这里可以做得很 粗暴 ，但你不想粗暴也没太好办法，这里很依赖模型能力就是了：

```
skill_cards = [{"name": s["name"], "desc": s["description"]} for s in skills]
prompt = f"""
你是技能路由器。根据用户问题，从技能列表中选择最相关的1-2个技能。
只输出JSON：{{"skills":["..."]}}，如果没有合适技能输出{{"skills":[]}}。
用户问题：{user_query}
技能列表：{skill_cards}
"""
r = openai_chat(model="gpt", messages=[{"role":"user","content":prompt}])
return parse_json(r)["skills"]
```

再接下来，就只需要将完整的skill目录加载即可：

```
blocks = []
for name in selected_names:
  s = find_skill(skills, name)
  blocks.append(f"""
# Skill: {s['name']} v{s['version']}
适用范围：{s['description']}
必须遵循的SOP：
{s['sop']}
""")
return "\n\n".join(blocks) or "你是通用企业助手，如无明确技能则保持谨慎，必要时追问。"
```

最后一步是只加载需要的Tools，这也是为什么我们说Skills是在做Tools的工程优化：

```
skills = load_skill_registry()
selected = route_skills(user_query, skills)
system_prompt = build_system_prompt(selected, skills)
allowed_tools = collect_tools_from_selected(selected, skills)  # 关键：只给相关工具
tools_schema  = build_openai_tools_schema(allowed_tools)
messages = [
  {"role":"system", "content": system_prompt},
  {"role":"user", "content": user_query}
]
while True:
  resp = openai_chat(model="gpt", messages=messages, tools=tools_schema)
if resp.is_tool_call:
    result = run_tool(resp.tool_name, resp.arguments)  # 外部执行
    messages.append(resp.as_assistant_message())       # 记录tool_call
    messages.append({"role":"tool","name":resp.tool_name,"content":json.dumps(result)})
    continue
return resp.content
```

到这里，你已经在 OpenAI 上**做出了一套 Claude Skills 的核心机制**...

Skills是工程优化

从最简伪系统其实就可以轻易看出来其中差别了：

没有 Skills 的典型 Agent

系统提示里塞一堆规则 + tools 全量注册 + 模型在一个巨大“工具目录 + 规则目录”里做选择

结果就是：乱调用/漏调用/乱序/延迟高/成本高/难维护...

如果有Skills呢

1. 先路由：把问题归类到少数 skill
2. 再加载：只把这个 skill 的 SOP 拉进上下文

3. 再裁剪：只把该 skill 需要的工具声明给模型

结果：工具更准、流程更稳、提示更可维护

这里的核心优化就是 从全量**Tools**加载到先用问题是筛选一次 ，当然也会存在筛选不准的情况（比如漏了）

流程稳定性问题

之前Tools的调用完全依赖与模型的推理规划能力，这个时候工具的使用就是个黑盒，Skills出现后，就直接在提示词里面写死了调用顺序、调用方式，稳定性肯定大大提高的。

只不过这里其实是将工作量或者复杂度转移出来了： **Agent**是以模型能力解决**Workflow**泛化能力不足的问题、**Skills**是以显性**SOP**增加工程工作量从而增加模型稳定性的策略 。

结语

综上，Skills确实是模型对Agent进行的工程层面的优化，他一方面提升工具的正确率（Function Calling），另一方面提供具体调用的稳定性（Workflow）。

只不过，该优化是建立在工具过多的场景，并且他也有一定局限性，比如100以内数量会比较好使，但如果数量增长到1000的话，新的工程问题又会出现：

> 之前是Tools多了，系统判断费劲；后续极有可能skill多了，其中也会出现类似的，依旧会出现之前的问题；这里包括各种版本结构造成的历史问题。

只不过真的到1000这个量级后，又会有相应的工程架构出现，了不起给Skills再包一层Skills嘛，就是测试起来费劲...

叶小钗

鹅厂、ctrip、baidu、一线开发，B站技术专家，独角兽技术负责人，AI产品项目负责...

149篇原创内容