从代码到知识：Agentic Loop 重塑企业 AI 应用

> 我们面对一个看似需要复杂程序的任务：将数千行电网稳规文档转换为可执行的 DSL 代码。
>
> 传统方案：写解析器、编规则、调边界、年年维护。
>
> 我们的选择：构建自主智能体基座，让 Agent 自己学习转换规则，自己验证生成结果，自己在错误中迭代改进。
>
> 这个选择背后，是一种正在浮现的开发范式——以 Agentic Loop 为基座，以知识为核心，以自主学习为灵魂。

引言：一个反直觉的选择

当面对"将稳规文档转换为 DSL 代码"这个任务时，传统思路是：

1. 分析文档结构，设计解析器
2. 编写转换规则，处理边界情况
3. 测试、调试、部署
4. 每年文档更新时，重复以上步骤

这是软件工程的标准路径。但我们做了一个反直觉的选择：**不写转换程序**。

我们的选择是：构建自主智能体（Agentic）基座，让 Agent 自己学习转换规则，自己验证生成结果，自己在错误中迭代改进。

因为我们意识到一个根本性问题：**知识被锁死在代码里**。当转换规则硬编码在代码逻辑中，改规则就是改代码。每一次规则变化都需要程序员介入。领域专家看不懂代码，程序员不懂领域知识，两者之间的鸿沟永远存在，传统转换程序解决不了这一困境。

2024-2025 年，LLM 的自主能力发生了质的飞跃——不再是"一问一答"的工具，而是能够自主迭代、自主验证、自主修正的执行引擎。这种能力的成熟，使得这一类复杂任务有了全新的解法。

本文将展开这个选择背后的思考。

第一部分：Agentic Loop——新范式的起点

什么是 Agentic Loop?

2024-2025 年，AI 领域最重要的进展不是模型参数的增长，而是 **Agent 能力的涌现**。

传统的 LLM 使用方式是"一问一答"：

```
用户：请帮我写一个排序函数
LLM: def sort(arr): return sorted(arr)
```

而 Agentic Loop 是"自主迭代直到完成"：

```
用户：请完成这个项目的单元测试
Agent:
  → 读取项目结构
  → 分析现有代码
  → 生成测试用例
  → 运行测试
  → 发现失败，分析原因
  → 修正测试代码
  → 再次运行
  → 全部通过，任务完成
```

关键区别在于：**Agent 不需要人类在每一步给出指令**。它自主决定下一步做什么，自主判断任务是否完成，自主修正错误。

这种能力的出现，使得一种全新的开发范式成为可能。

为什么 Agentic Loop 可以成为开发基座?

传统程序的执行是**确定性**的：相同输入必然产生相同输出。这种确定性来自于代码逻辑的固定性。

Agentic Loop 的执行是**目标导向**的：给定目标，Agent 自主寻找达成路径。路径可能不同，但目标是确定的。

这种差异带来了根本性的变化：

维度	传统程序	Agentic Loop
执行逻辑	预先编写，固定不变	实时推理，动态调整
错误处理	预定义异常分支	自主分析，自主修正
知识来源	硬编码在代码中	从外部知识库读取
扩展方式	修改代码，重新部署	更新知识，立即生效
调试方式	断点、日志、堆栈	对话、解释、推理

当我们把 Agentic Loop 作为开发基座，我们实际上是在说：**让 LLM 的推理能力替代固定的代码逻辑**。

完成条件：Agentic Loop 的灵魂

一个 Agentic Loop 能否可靠运行，取决于**完成条件的设计**。

完成条件必须满足两个要求：

1. **可观测**：Agent 能够自主验证条件是否满足
2. **明确**：不存在模糊地带

以 MD→DSL 转换为例，我们的完成条件是：

```
1. 语法正确：生成的 DSL 能被解析器接受
2. 变量存在：所有 $(变量) 在全局定义中存在
3. 结构完整：IF-ENDIF 配对，表格标签闭合
4. 语义等价：与源文档的对应关系正确
```

每一个条件，Agent 都可以自主验证：调用解析器、搜索变量定义、计算配对数量、对照源文档。

当所有条件满足，Agent 输出完成标志；否则，继续迭代修正。

这就是 Agentic Loop 的运行机制：**目标驱动，自主迭代，条件收敛**。

第二部分：Skills 与 Commands——知识的载体

从函数到 Skill：一次认知跃迁

传统编程中，我们用**函数**封装可复用的逻辑：

```python
def convert_table(md_table: str) -> str:
    # 100 行转换逻辑
    return dsl_table
```

函数的问题在于：它是给机器执行的，不是给人理解的。领域专家看不懂 `re.findall(r'\|([^|]+)\|', row)`，程序员也不一定理解为什么要这样处理。

Skill 是给 LLM 理解的任务描述：

```markdown
# wengui-retrieval

支持对稳规文档的结构化访问：
- 获取目录结构
- 获取章节内容（支持递归获取子章节）
- 关键词搜索
- 表格提取
- 引用解析

默认返回 Markdown 原文，不添加任何注释或元信息。
```

Skill 不是代码，而是**意图的声明**。它告诉 LLM：这个工具能做什么，应该在什么场景下使用，输出是什么形式。

LLM 根据 Skill 的描述，自主决定如何调用、如何组合、如何处理结果。

Skill 的三层架构

在我们的系统中，Skills 形成了清晰的三层架构：

```
┌─────────────────────────────────────────────────────────┐
│  Layer 1: 数据获取层                                    │
│  ├── wengui-retrieval: 访问 MD 文档                     │
│  └── wengui-dsl-retrieval: 访问 DSL 代码                │
├─────────────────────────────────────────────────────────┤
│  Layer 2: 知识层                                        │
│  └── knowledge-retrieval: 访问知识库                    │
│      ├── 元知识（语法规则）                             │
│      ├── 领域知识（转换模式）                           │
│      └── 样本库（配对示例）                             │
├─────────────────────────────────────────────────────────┤
│  Layer 3: 执行层                                        │
│  └── Claude Code 内置能力                               │
│      ├── 文件读写                                       │
```

```
15 │       └── LLM 推理                                                    │
16 └──────────────────────────────────────────────────────────────────────┘
```

每一层都是**声明式**的：描述能力，而非实现细节。LLM 在执行任务时，自主决定调用哪些 Skills，以什么顺序，如何组合结果。

Commands：工作流的入口

如果说 Skill 是能力的声明，那么 Command 就是**工作流的入口**。

```
# /md2dsl

将稳规 MD 章节转换为 DSL 格式。

## 执行步骤
1. 读取 MD 章节内容
2. 获取转换知识
3. 生成 DSL 代码
4. 验证并迭代修正
5. 输出完成标志

## 完成条件
- 语法验证通过
- 变量引用正确
- 结构完整
```

Command 定义了一个完整的任务流程，但**不规定具体实现**。LLM 根据 Command 的描述，自主编排 Skills 的调用，自主处理中间状态，自主判断完成条件。

这就是 Agentic 开发范式的核心：**我们描述"做什么"，LLM 决定"怎么做"**。

第三部分：知识的自主学习——新范式的灵魂

知识从哪里来?

这是整个范式中最关键的问题。

传统方式：**人工编写知识**

- 领域专家总结规则
- 程序员翻译成代码或配置
- 每次更新都需要人工介入

我们的方式：**Agent 自主学习知识**

这不是一句口号，而是一个可实现的机制。

学习的本质：从样本中提取可泛化的模式

考虑这样一个学习过程：

```
输入：章节 6.1.1 的 MD 原文和对应的 DSL 代码
输出：可用于转换其他章节的知识

Agent 的学习过程：
1. 对比 MD 和 DSL，识别对应关系
2. 抽象出转换模式（不是具体数值，而是规则）
3. 验证模式的泛化能力（能否用于其他章节）
4. 存储到知识库
```

关键在于第 3 步：**验证泛化能力**。

如果学到的只是"300 → <3000"这样的具体映射，那是**记忆**，不是**知识**。
如果学到的是"数值×10，万千瓦→MW"这样的转换规则，那才是**知识**。

防止"抄袭"：交叉验证机制

一个微妙但关键的问题：如果用章节 A 的样本来指导生成章节 A 的 DSL，本质上是"开卷考试"。

我们的解决方案：**语义检索 + 排除同章节**

```
当转换章节 6.1.1 时：
1. 从知识库检索相似样本
2. 排除章节 6.1.1 自身的样本
3. 使用其他相似章节的样本作为参考
```

这确保了：Agent 必须真正理解转换规则，而不是简单复制。

更进一步，我们实现了 **K-折交叉验证**：

```
将所有样本分成 K 份
对于每一份：
  - 用其他 K-1 份的知识
  - 生成这一份的 DSL
  - 与原始 DSL 对比
  - 计算准确率

如果准确率高，说明学到的是可泛化的知识
如果准确率低，说明只是记忆了特定映射
```

这是机器学习中验证模型泛化能力的标准方法，我们将其应用于知识质量的评估。

知识的分层存储

学到的知识需要有效组织。我们采用三层结构：

```
knowledge/
├── meta/            # 元知识：语法规则
│   ├── 单位转换规则
│   ├── 符号使用规则
│   └── 命名规范
│
├── domain/          # 领域知识：可泛化模式
│   ├── 表格结构模式
│   ├── 条件嵌套规则
│   └── 隐含公式模板
│
└── samples/         # 样本库：按特征分类
    ├── 按表格类型/
    ├── 按章节类型/
    └── 按复杂度/
```

注意：样本不是按章节号分类，而是**按特征分类**。这确保了检索时能找到真正相似的样本，而不是同一章节的"答案"。

增量学习：知识的持续演进

知识不是一次性获取的，而是持续积累的。

```
当 Agent 成功完成一次转换：
1. 分析这次转换中使用的模式
2. 检查是否有新的模式
3. 验证新模式的泛化能力
4. 如果验证通过，添加到知识库
5. 记录版本，支持回溯
```

这意味着：**系统在使用中自动变得更强**。

每一次成功的转换，都可能贡献新的知识。每一次失败的转换，都暴露知识的缺陷，触发学习。

这就是自主知识学习的完整图景：

- **学习**：从样本中提取可泛化的模式
- **验证**：通过交叉验证确保是"知识"而非"记忆"
- **组织**：分层存储，按特征检索
- **演进**：增量学习，持续改进

第四部分：边界与可能性

这个范式适合什么场景?

Agentic 开发范式特别适合：

1. **规则复杂但可描述**的任务
    - 文档转换、格式处理
    - 规则引擎、决策系统
    - 领域特定语言处理
2. **知识密集型**的任务
    - 需要大量领域知识
    - 规则经常变化
    - 边界情况多
3. **需要人机协作**的任务
    - 领域专家参与
    - 需要解释和审核
    - 迭代改进

边界在哪里?

Agentic 范式不适合：

1. **性能关键**的场景——LLM 推理比确定性代码慢，实时系统、高频交易不适用
2. **确定性要求极高**的场景——相同输入必须产生完全相同的输出，安全关键系统需谨慎
3. **知识难以外化**的场景——隐性知识、直觉判断，需要大量上下文的决策

正在发生的变化

当 LLM 能力继续提升，Agentic 范式的边界会不断扩展。

我们已经看到：

- **嵌套的 Agentic Loop**：Agent 在执行中发现知识缺陷，自动触发学习循环
- **多 Agent 协作**：不同专长的 Agent 协同完成复杂任务
- **跨项目知识迁移**：在一个项目中学到的知识，自动应用于相似项目

这不是科幻，而是正在发生的技术演进。

第五部分：当开发范式改变，更多的改变随之而来

当我们真正用 Agentic 范式构建系统时，一些更深层的变化开始浮现。

知识的存在形态变了

这里需要澄清一个误解：我们不是"不写程序"——程序员仍然要构建大量的系统级底层支撑：Agentic Loop 基座、Skills 框架、验证机制、完成条件检测...这些是自主智能体的骨架。

真正变化的是：**企业领域知识的存在形态**。

传统方式下，领域规则被硬编码在代码逻辑中。"当开机台数小于 3 时，限额为 800"——这句话变成了 `if count < 3: limit = 800`。规则和代码紧紧绑定，改规则就是改代码。

在 Agentic 范式下，同样的规则存在于知识文档中。自主智能体读取知识、理解知识、应用知识。规则变了，更新文档即可，基座不需要改动。

这是工作重心的转移：程序员专注于构建强大的自主智能体基座，企业知识则成为可独立管理、可持续扩展的资产。

基座 + 知识 = 跨越式的执行能力

自主智能体基座本身是一个"通用执行引擎"——它懂得如何迭代、如何验证、如何学习，但不懂任何特定领域。

知识资产是"领域智慧"——它包含转换规则、边界情况、隐含模式，但自己不能执行任何事情。

当两者结合：基座获得了领域智慧，知识获得了执行能力。

这种组合产生的能力是跨越式的。不是 1+1=2，而是涌现出单独任何一方都不具备的能力：自主处理从未见过的情况、从错误中学习新模式、在使用中持续变强。

业务专家走到台前

传统模式中，业务专家的知识要经过漫长的"翻译链条"才能进入系统：

```
业务专家 → 需求文档 → 产品经理 → 设计文档 → 程序员 → 代码
```

每一次翻译都有信息损失。

Agentic 范式缩短了这个链条。领域专家可以用更接近自然语言的方式描述规则，自主智能体直接理解和执行。

这不是说程序员不再需要——而是分工更清晰了：程序员构建和优化自主智能体基座，业务专家贡献和维护领域知识。两者各司其职，又紧密协作。

持续演进成为常态

传统软件有清晰的"版本"概念。1.0 上线，运行一年，需求积累够了，启动 2.0 项目。

在 Agentic 范式下，"版本"的边界变得模糊。

每一次成功的执行，都可能贡献一条新的知识。每一次失败，都暴露知识的缺陷，触发学习。知识资产在使用中自动增长，基座上的执行能力随之增强。

没有人专门启动了"升级项目"，但系统确实变得更强了。

这是一种新的常态：企业 AI 应用从"建设-维护"的二元模式，转向"持续演进"的一元模式。

结语：一种正在发生的变化

回顾我们的选择：面对一个复杂的转换任务，我们没有写转换程序，而是构建了自主智能体基座，让 Agent 自己学习、验证、迭代。

这个选择之所以可行，是因为 Agentic 范式已经成熟到可以承担这类任务。而当我们真正用这种方式构建系统时，一些连锁反应开始显现：

开发范式变了——从编写固定逻辑，到定义目标和完成条件。

知识的存在形态变了——从硬编码在代码中，到独立存在于可读写的文档里。

企业 AI 应用的建设方式也在变——从"建设-维护"的项目制，到"持续演进"的常态化。

这不是预测，而是正在发生的事情。

对程序员而言，这意味着新的技能要求：设计知识结构、定义验证条件、与 Agent 协作调试。
对企业而言，这意味着新的可能性：领域知识成为可独立管理的资产，系统在使用中自动变强。

我们只是在一个具体项目中做了一个具体选择。但这个选择背后的逻辑，或许值得更多人思考。

附录

核心概念澄清

概念	定义	负责方
自主智能体基座	Agentic Loop、Skills、验证机制等底层支撑	程序员构建
知识资产	企业领域知识的可管理、可扩展形态	业务专家+程序员共同维护
执行能力	自主智能体基于知识资产产生的强大能力	基座+知识 = 跨越式能力

相关文档

文档	说明
MD2DSL_DESIGN.md	MD→DSL 转换的具体实现方案
KNOWLEDGE_LEARNING_DESIGN.md	知识学习系统的详细设计

核心组件清单

组件	类型	说明
wengui-retrieval	Skill	MD 文档结构化访问
wengui-dsl-retrieval	Skill	DSL 代码智能提取
knowledge-retrieval	Skill	知识库检索与学习
/learn-patterns	Command	知识积累工作流
/md2dsl	Command	转换执行工作流

术语表

术语	定义
Agentic Loop	LLM 自主迭代执行任务直到完成的机制
Skill	给 LLM 理解的能力声明，描述工具的用途和使用方式
Command	工作流的入口，定义任务流程和完成条件
交叉验证	评估知识泛化能力的方法，确保学到的是"知识"而非"记忆"
增量学习	系统在使用中自动积累新知识的机制

> **文档版本**: 3.0
> **最后更新**: 2026-01-30