Spell：使用大语言模型合成程序化编辑

Daniel Ramos，卡内基梅隆大学和 INESC-ID / IST - 里斯本大学，葡萄牙Catarina Gamboa，卡内基梅隆大学和 LASIGE / FCUL - 里斯本大学，葡萄牙Inês Lynce，INESC-ID / IST - 里斯本大学，葡萄牙Vasco Manquinho，INESC-ID / IST - 里斯本大学，葡萄牙Ruben Martins，卡内基梅隆大学，美国Claire Le Goues，卡内基梅隆大学，美国

arXiv:2602.01107v1 [cs.SE] 1 Feb 2026

库迁移是软件开发中常见但容易出错的任务。开发者可能因需求变化或许可变更等原因，需要用一个库替换另一个库。迁移通常需要手动更新和重写源代码。虽然已有自动化迁移工具，但大多数工具依赖从已经完成类似迁移的真实项目中挖掘示例。然而，这类数据十分稀缺，而且很难为任意库对收集此类数据。此外，这些迁移工具往往未能充分利用现代代码转换基础设施。

本文提出了一种自动化 API 迁移的新方法，避开了上述局限。我们不依赖现有迁移数据，也不直接使用大语言模型进行转换，而是利用大语言模型提取迁移示例。接着，我们使用智能体将这些示例泛化为可复用的转换脚本，并在 `PolyglotPiranha` 这一现代代码转换工具中实现。我们的方法将大语言模型中的潜在迁移知识提炼为结构化、可测试且可重复的迁移逻辑，无需预先准备语料库，也无需投入人工工程工作量。针对 `Python` 个库的实验结果表明，我们的系统能够生成多样的迁移示例，并合成可泛化至真实代码库的转换脚本。

1 引言

库是现代软件开发的基石。它们通过应用程序接口（API）提供模块化功能 [10], ，这些接口在暴露功能的同时抽象了实现细节。库迁移是指将客户端代码库中使用的一个软件库替换为另一个软件库的过程。此类迁移对于适应不断变化的软件需求十分必要，例如需要新功能、许可证变更，或更现代、支持更完善的 API [5, 7, 12, 37, 50]。将原始库的 API 调用替换为目标库的 API 调用，很大程度上仍是一项人工 [28]且容易出错的 [27] 任务。

库迁移普遍存在且代价高昂，这促使人们研究自动化方法。现有的大多数方法都挖掘已经完成迁移的软件代码仓库 [15–17, 26, 31, 34, 43, 51],，将示例归纳并系统化为可复用的格式。然而，这些技术通常受限于对历史迁移数据的依赖，而这类数据在实践中既难以获取 [29]，质量也参差不齐。

与此同时，大语言模型（LLM）已展现出潜力，并在代码变换辅助方面日益受到关注，包括在工业场景中的应用（例如 Amazon Q Transform [42]）。然而，直接使用大语言模型进行迁移往往不可靠，可能引入不可预测的变更、格式或风格上的变更，或细微的语义错误 [22]。对于相似的迁移模式，它们的表现也可能不一致，且不会提供可复用或可检查的产物，供开发者了解已完成的工作。最后，基于大语言模型的迁移必须低效地从头生成大部分变更。

作者联系方式：Daniel Ramos，danielrr@cmu.edu，卡内基梅隆大学和 INESC-ID / IST - 里斯本大学，葡萄牙；Catarina Gamboa，cgamboa@andrew.cmu.edu，卡内基梅隆大学和 LASIGE / FCUL - 里斯本大学，葡萄牙；Inês Lynce，ines.lynce@inesc-id.pt，INESC-ID / IST - 里斯本大学，葡萄牙；Vasco Manquinho，vasco.maquinho@inesc-id.pt，INESC-ID / IST - 里斯本大学，葡萄牙；RubenMartins，rubenm@cs.cmu.edu，卡内基梅隆大学，美国；Claire Le Goues，clegoues@cs.cmu.edu，卡内基梅隆大学，美国。

，第1卷，第1期，文章。出版日期：2026年2月。

(a) 之前（使用 `cryptography.fernet`）

```python
import hashlib
from cryptography.fernet import Fernet
(...)
def encrypt_document(document: str, key: bytes) -> bytes:
    cipher = Fernet(key)
    encrypted = cipher.encrypt(document.encode())
    return encrypted
(...)
```

(b) 之后（使用 `pycryptodome`）

```python
from Crypto.Cipher import AES
from Crypto.Util.Padding import pad
(...)
def encrypt_document(document: str, key: bytes) -> bytes:
    cipher = AES.new(key, AES.MODE_CBC)
    padded_data = pad(document.encode(), AES.block_size)
    encrypted = iv + cipher.encrypt(padded_data)
    return encrypted
```

(c) 表达该转换的合成 `PolyglotPiranha` 程序

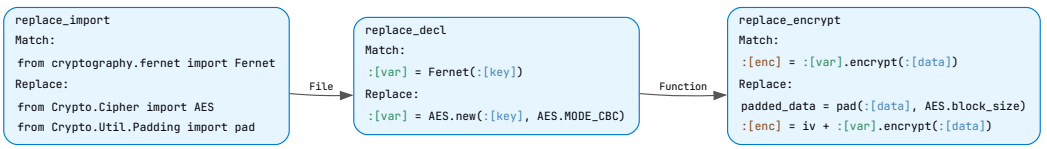

图 1. 从 `cryptography.fernet` 迁移到 `pycryptodome`：迁移前后的函数级加密逻辑，以及从该示例中提取的相应 `PolyglotPiranha` 转换脚本。

本文介绍了 Spell (**S**ynthesis of **P**rogrammatic **E**dits using **LL**Ms），它不将大语言模型用作黑盒代码翻译器，而是将其用作知识源，以系统地生成已验证的迁移示例，再将这些示例提炼为精确且可复用的转换脚本。通过在大规模代码语料上进行预训练，大语言模型能够学习 API 及其语义关系（包括跨库映射）之间的联合表示 [41]。也就是说，大语言模型隐式编码了有关 API 关系和使用模式的丰富知识。这些知识可能并不完美，或者噪声过多，无法直接应用。然而，我们的关键洞见是，可以系统地提取并验证这些知识，以支持迁移脚本合成。这种方法结合了大语言模型所编码的广泛 API 知识，以及基于规则的转换所具备的可靠性和效率，且无需历史迁移数据。

Spell 的一项重要优势是，它会生成用于代码变换的领域特定语言（DSL）迁移脚本（具体来说是 `PolyglotPiranha` [25]）。虽然大多数既有工作会生成或依赖自定义的转换工具集 [15–17, 26],，Spell 旨在利用现代 DSL 和框架，例如 OpenRewrite [9],、GritQL [45],和 `PolyglotPiranha` [25]，它们支持在大型代码库中执行自动化且一致的转换。因此，Spell 的输出可作为可维护的资产，供开发者检查、进行版本控制、调试和优化。这些 DSL 日益融入 Uber [25] 和亚马逊 [42], 等大型科技公司的 CI/CD流水线，使我们的方法能够自然地融入现代工作流程。总体而言，这一用例既能帮助在发布弃用变更时为客户端代码库提供支持的库维护者，也能帮助缺乏可供挖掘的历史数据、需要迁移代码的客户端团队。

Spell 采用一种新颖的混合方法，根据生成的示例合成 `PolyglotPiranha`语言中的迁移脚本。Spell 首先采用经典程序综合方法进行基于反合一的合成。这一步会确定性地从具体代码差异中推断语法模式，生成一组初始的局部转换规则，用以表达迁移的部分内容。然而，`PolyglotPiranha`与其他转换框架和领域专用语言一样， [9, 32, 45], 将迁移表示为

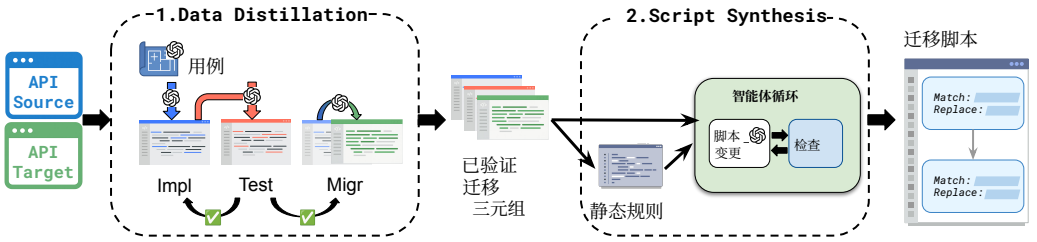

图 2. Spell 流水线概览。给定一对源 API 和目标 API，阶段 1 将大语言模型中的数据提炼为用例。这些用例指导示例的生成，示例包含源 API 中的实现、迁移到目标 API 的过程，以及用于验证这两种实现的测试。阶段 2 使用一组初始的静态推断规则和智能体循环来更新规则，并合成迁移脚本。最终输出是一个项目无关迁移脚本，可自动应用于将源 API 迁移到目标 API。

局部作用域规则的组合，以捕捉语义关系、依赖、链式组合或作用域。因此，Spell 将经典程序综合与智能体工作流相结合，以细化、组合并协调最初推断出的局部规则，形成完整的 `PolyglotPiranha` 程序，充分发挥现代变换DSL的表达能力。

我们的结果表明，Spell 能够针对各种 Python 库迁移任务合成正确且可复用的迁移脚本。在九项迁移任务中，它为每项任务平均生成了 87 个已验证示例，并在单次试验中为 61.6% 的任务推断出正确脚本。Spell 在每项任务上的表现都优于 Melt（一种最先进的已有方法）。此外，这些脚本能够泛化到从开源仓库中选取的真实项目。我们刻意将所有实验的大语言模型预算控制在较低水平（< $100），证明使用小模型也能以较低成本获得这些结果。这些发现表明，可以从大语言模型中提炼结构化迁移知识，并将其转化为可复用的迁移脚本。

总之，我们做出了以下贡献：
(1) 一种新颖的方法，可在现有的转换领域专用语言中自动生成可复用、已验证的迁移脚本。(2) 一种数据生成方法，可系统地从大语言模型中提取并验证结构化迁移知识，而无需历史迁移日志。(3) 一种新颖的智能体式工作流，利用经典程序综合将示例转换为可复用的迁移脚本。(4) 一个新的数据集，包含 870 个已验证迁移示例及其对应测试用例，涵盖十项Python 迁移任务。(5) 一个实现，Spell，其一次性脚本综合正确率达到 61.6%，并在每项评估任务上的表现均优于此前最先进的方法。我们展示了这些脚本在真实开源项目中的应用。(6) 一个开源产物，其中包含Spell 实现、数据集、评估工具链，以及所有合成的脚本和结果，均已公开提供 [38]。

2 示意性示例

我们的流程分为两个阶段：（1）数据蒸馏，生成包含测试的代码迁移示例；（2）合成，将这些成对内容抽象为可复用的迁移脚本。图 2展示了整体流程。

数据蒸馏。数据生成的目标是以示例的形式从大语言模型中提炼代码迁移知识。举例来说，考虑在两个 Python cryptography 库之间进行迁移的任务： `cryptography`[1] 和 `pycryptodome`。[2] 给定一对库（源库，目标库），我们的方法使用大语言模型生成同一功能的两个实现，分别使用其中一个库。这些成对的实现构成迁移示例的基础，之后会进一步将其泛化。

第一步是让大语言模型生成用例，即可能通过任一库实现的任务或功能。这些用例有助于让模型聚焦于一个可通过两个 API 实现的共同功能目标。当要求针对 `cryptography` 和`pycryptodome`提出这样的用例时， `gpt4o-mini`回答：

> 可以使用 *pycryptodome* 或 *cryptography* 实现文件加密工具，让用户能够加密和解密文本文档。高级行为和 *API*：（.....）

然后，我们提示模型使用源库为此用例生成实现。图1a 展示了大语言模型使用 `cryptography`库生成的一个实现。我们还提示大语言模型为每个实现生成测试以验证功能，并过滤掉明显不真实的示例。

最后，我们提示大语言模型将每个实现从 `cryptography` 迁移到替代库 `pycryptodome`。图 1b 展示了将图1a中的代码迁移后得到的结果。迁移提示要求大语言模型仅修改特定于库的 API 调用，其余代码保持不变。这样便可将为原始实现生成的测试复用于迁移后的版本。如果迁移后的代码通过测试验证的行为与原始代码相同，测试就能表明迁移成功，因此也可用于筛选示例。

该过程会生成一组代码迁移示例，由某个用例的实现对组成，并共享测试文件。第 3 节将详细介绍数据生成。

合成迁移脚本。 第二阶段将合成迁移示例抽象为可复用的迁移程序，这些程序能够捕获结构化转换模式，并可应用于类似任务或大型代码库。我们使用 `PolyglotPiranha` [25]语言生成程序，该语言是一种用于源到源转换的领域专用语言。 `PolyglotPiranha` 程序由`rules`组成，每个`rules`都包含一个 `match` 子句，用于识别代码模式，以及一个 `replace`子句，用于重写该模式。模板变量（例如 `:[var]`）使规则能够针对特定名称和表达式进行泛化，其作用类似于正则表达式，但面向语法感知代码模式。规则可以通过带标签的边连接，这些边规定了应如何触发后续转换。这样便可实现级联式和上下文敏感重写。

图 1c展示了一个 `PolyglotPiranha` 根据该示例合成的程序。第一条规则`replace import`将原始导入替换为目标库中的导入。这会在同一文件中触发第二条规则 `replace decl`，后者重写对象构造过程。第三条规则 `replace encrypt`更新加密 API 的使用方式。模板变量 `:[var]` 和 `:[data]` 使这些规则能够泛化到共享相同结构、但使用不同变量名称或具体表达式的不同程序。

此步骤的输出是一组可解释的迁移程序，它们捕捉了转换的本质，并可应用于或扩展到类似场景。第4 节详细介绍如何从合成示例中进行脚本合成。

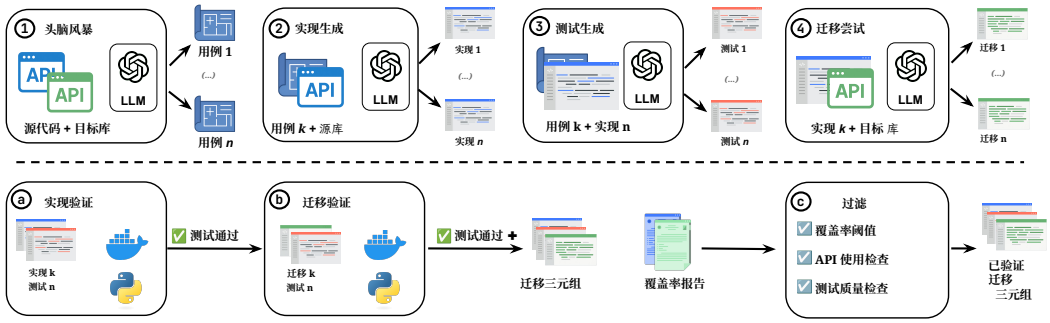

图 3. 使用大语言模型生成迁移示例的流程。给定一个源–目标库对，我们提示大语言模型提出抽象的用例，利用对两个库的潜在知识（步骤 1）。针对每个用例，我们生成多个源库实现（步骤 2）和测试套件（步骤 3），并将其迁移到目标库（步骤 4）。验证包括实现验证（a）、迁移验证（b）和质量过滤（c）。最终输出是一组已验证的迁移三元组（实现、测试、迁移）。

3 迁移数据蒸馏

图3 概述了数据蒸馏过程，该过程提取并验证大语言模型关于两个库之间迁移的预训练知识。此过程利用源库与目标库生成多组等价程序，并通过测试用例为其功能等价性提供证据。这些数据是迁移脚本合成（第4节）的输入。

更准确地说：给定源库 S 和目标库 T，我们从大语言模型中采样迁移知识，得到原始迁移三元组 $\mathrm{M} = \{(\tilde{s}_i, \tilde{t}_i, \tilde{m}_i)\}_{i=1}^{n}$。这些三元组由自动生成，可能包含冗余、不一致或错误。我们对该集合进行验证和过滤，以生成高质量子集 $\mathrm{M} = (s_i, t_i, m_i)_{i=1}^{m}$，其中 $m \le n$，并且：

- s_i 是使用 S 的实现
- m_i 是使用 T 的功能等价实现
- t_i 是一个可在 s_i 和 m_i 上执行并通过的测试套件

我们使用 $\sim$ 来表示可能有效也可能无效的采样三元组；不带 $\sim$ 的记号指已验证的数据/三元组。我们现在解释知识抽取（第3.1节）和验证（第3.2节）。提示词因篇幅所限而省略，但可在我们的工件 [38]。

3.1 模型采样

模型采样的目标是生成共享功能示例实现的多样化初始集合，使用 $\mathcal{S}$ 和 $\mathcal{T}$。一个非平凡的库通常可以以多种方式使用。因此，我们的流程包含一个多阶段流水线，首先生成多个用例，即可以使用$\mathcal{S}$ 实现的功能（第3.1.1节），并分别生成实现，即使用$\mathcal{S}$ 实现这些用例（第3.1.2节）。对于每个用例的每个实现，该流水线会生成测试（第3.1.3节），最后将实现迁移为使用$\mathcal{T}$（第3.1.4节）。

3.1.1 用例生成。 给定库 $(\mathcal{S}, \mathcal{T})$，Spell 首先会提示大语言模型生成多个用例 C (图 3，步骤 1)。每个用例 $C_j \in C$ 代表一个可以使用任一库实现的用例、程序或其他有用功能。

[1]https://github.com/pyca/cryptography [2] https://github.com/Legrandin/pycryptodome

先生成用例，可以将头脑风暴迁移场景这一创造性任务与编写正确代码这一技术性任务分离开来。这有助于鼓励模型探索广泛的 API 用法，而不局限于常见的样板代码。它还提供了明确且共享的语义目标，促使源实现和目标实现保持一致。最后，对于有前景的用例，它允许流水线根据需要重试实现生成，而无需重新探索整个概念空间。分离这两项任务还可以让模型在投入计算资源进行实现之前，先专注于头脑风暴那些不复杂或不适合实现的用例。

用例生成首先利用模型创建一小组种子用例，随后通过自指令循环生成更多迁移场景。自指令是指模型利用种子数据生成自己的任务演示，已被证明能够在各种情境下有效激发模型已有的知识 [49]。我们首先通过直接向模型提供一条基本指令来生成 p 种子用例。每个用例都描述一个涉及源库的用例，并对其功能进行高层次概述。我们将这些种子用例收集到一个列表中，并在后续提示中将其作为少样本示例，以生成更多用例。

用例生成循环通过随机抽取 k 之前的用例（种子用例或生成的用例），将其纳入新的提示中，并要求模型提出一个新的用例。该过程会重复进行，直到生成目标数量的用例，其中每个用例都展示源库与目标库的一种使用方式。

3.1.2 实现生成。对于每个用例 $C_j \in C$，我们提示模型生成一组实现 $S_j = \{\tilde{s}_{j,1}, \tilde{s}_{j,2}, \ldots \tilde{s}_{j,n}\}$，使用源库$\mathcal{S}$(图3，步骤 2）。理想情况下，这些实现具有模块化结构，并提供定义明确的 API。这有助于迁移到$\mathcal{T}$，同时保留高层接口，从而能够在两者上使用相同的测试。

因此，提示要求模型在一个自包含的 `Python` 文件中实现该用例，并遵循以下约束：(1) 将用例 C_j 的 API 实现为一组函数；(2) 专注于 API 实现；可以选择添加一个小型 `main` 代码块；(3) 避免使用指定库 S 之外的外部依赖。

我们构造了一个多轮对话提示，其中包含原始用例和实现指令。为获得多样化的输出并提高模型生成可用实现的概率，我们使用随机解码对多个补全结果进行采样。

T此步骤的输出是针对每个 $C_j \in$ C 的一组最多包含 n 个潜在实现 。

3.1.3 测试生成。 对于每个用例-实现对（$C_j, \tilde{s}_{j,k}$），我们会生成 p 测试套件$\widetilde{T}_{j,k} = \{\tilde{t}_{j,k,1}, \tilde{t}_{j,k,2}, \ldots, \tilde{t}_{j,k,p}\}$（图 3，步骤 3）。每个测试套件 $\tilde{t}_{j,k,l}$旨在部分验证实现 $\tilde{s}_{j,k}$ 的正确性，并增强对其正确性的信心（之后也将用于验证使用目标库 T 迁移后的实现）。

理想的测试应针对实现的公共 API 和核心功能，而不是那些可能无法迁移的内部实现细节。例如，对于一个用于加密文件的程序，测试不应直接检查加密后的输出，而应验证解密文件是否能还原原始内容（以及其他功能）。提示会指示模型生成单元测试（如果无法进行单元测试，则生成端到端测试），以验证整个系统的行为。

提示包含 $C_j, \tilde{s}_{j,k}$以及测试生成指令。我们针对每个实现采样多个测试文件，以缓解模型幻觉以及测试生成中普遍存在的挑战，例如断言错误。采样能提高为原本正确的实现至少获得一个有效测试文件的概率。由于我们的实现专用于 `Python`，提示

会指示模型在适当情况下使用 pytest 测试框架，或使用hypothesis进行基于属性的测试。这一假设可以轻松移植到其他语言。

此步骤的输出是每个实现对应的多个测试文件。

3.1.4 迁移。对于每个用例—实现对（$C_j, \tilde{s}_{j,k}$），我们提示模型生成~多个迁移后的实现 $M_{j,k} = \{\tilde{m}_{j,k,1}, \tilde{m}_{j,k,2}, ..., \tilde{m}_{j,k,q}\}$，并使用目标库 T（图 3，步骤 4）。每个 $\tilde{m}_{j,k,r}$ 都代表一次迁移尝试，旨在迁移 $\tilde{s}_{j,k}$，同时保留其公共 API。这需要使用多轮对话提示，其中包含原始用例 C_j 和实现 $\tilde{s}_{j,k}$，并指示模型将源实现迁移为使用目标库。提示还要求模型保持相同的结构和函数签名，仅更新内部逻辑和 API 调用，使其使用目标库。

~ ~ ~此步骤的输出是原始迁移三元组集合 $\mathcal{M} = \bigcup_{j,k} \{\tilde{s}_{j,k}\} \times T_{j,k} \times M_{j,k}$，其中~对所有用例取并集 $C_j \in C$ 及其实现。 $\mathcal{M}$ 可能包含错误或仅部分迁移的示例；这些示例会按照第 3.2 节所述进行验证和筛选。

3.2 验证

并非所有采样的迁移示例都是有效的。初始实现或迁移后的实现可能是虚假的~或不完整，测试可能无法运行，等等。因此，验证步骤会过滤 M，以生成一组有效迁移三元组 $\mathrm{M} = \{(s_i, t_i, m_i)\}_{i=1}^{m}$。在有效的三元组中，源实现与迁移后的实现都会暴露所需的 API，通过相同的测试，并以足够的测试覆盖率得到执行。

实现验证。 首先，我们验证初始用例实现。对于每个 $(\tilde{s}_{j,k}, \tilde{t}_{j,k,\ell}) \in \widetilde{S}_j \times \widetilde{T}_{j,k}$，我们将测试 $\widetilde{T}_{j,k}$ 针对 $\widetilde{s}_{j,k}$ 执行，并保留所有测试均通过的配对。回想一下，我们会为每个实现生成多个潜在的测试套件（第 3.1.3 节）；因此，此步骤可能产生多个包含给定 $s_{j,k}$ 的配对。我们独立运行每个测试文件 $\tilde{t}_{j,k,\ell}$，使用 pytest，在一个 docker 容器中进行，该容器包含锁定的 Python 版本以及大量Python 库，包括 S 及其依赖。

迁移验证。 对于每个有效的 $(s_{j,k}, t_{j,k,\ell})$ 三元组及其关联的迁移尝试 $\tilde{m}_{j,k,r} \in M_{j,k}$，我们通过确认 $\tilde{m}_{j,k,r}$ 也能通过测试 $t_{j,k,\ell}$（舍弃未通过者），来提高对等价性的信心。由于预期迁移后的代码会保留原始 API，因此预期同一测试套件仍然适用（不适用的将被舍弃）。

基于测试的过滤。 最后，我们还会收集每次运行的行级覆盖率指标，以建立对测试套件能够有效执行核心逻辑的信心。同时，我们收集有关 API 使用情况的信息，以供后续分析。我们保留共享测试在两个实现中均达到至少 60% 行覆盖率的三元组。

为了构建最终已验证的 M：对于每个用例 $\tilde{s}_{j,k}$ 中属于多个有效三元组的实现 C_j，我们选择迁移后测试覆盖率最高的三元组。仅属于单个有效三元组的实现则直接纳入。所选三元组构成 $(s_i, t_i, m_i) \in \mathrm{M}$。因此，每个已验证的实现最多向 M 贡献一个三元组，在确保示例高质量的同时，保持用例实现多样性。

4 迁移脚本合成

Spell将迁移三元组 $\mathcal{M} = \{(s_i, t_i, m_i)\}_{i=1}^{m}$（第3节）转换为可执行重写脚本并写入 PolyglotPiranha，即一套用于代码变换的最先进水平领域专用语言与工具集（第4.1节）。我们用于脚本推断的新方法将经典反合一与智能体大语言模型策略相结合。这种混合方法源于 PolyglotPiranha 程序的结构：局部重写规则（图节点）由高层策略（带标签的边）连接。经典反合一

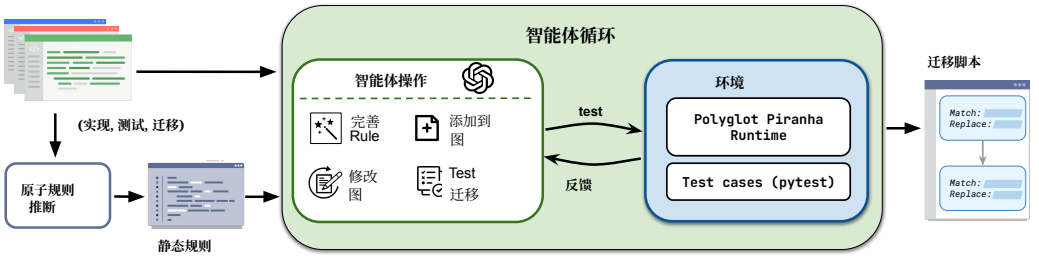

图 4. Spell 的迁移脚本合成方法所采用的智能体式工作流。我们的流程利用原子规则推断算法创建一组初始规则，随后将其交由大语言模型智能体进行 `Polyglot-Piranha` 脚本生成。该智能体会迭代地对脚本应用变更，并根据测试和 `PolyglotPiranha`的运行时反馈来优化规则。

擅长从具体示例中推断出精确的局部规则，但无法推断编排策略、作用范围或级联效应。因此，我们以基于大语言模型的技术补充传统合成，将机械推断得到的原子规则组织成连贯的转换策略。

对于每个迁移三元组 $(s, t, m) \in \mathcal{M}$，脚本合成包括：(1) **规则推断**：从源目标差异代码块自动生成初始低级转换规则（第 4.2节）；以及 (2)**基于智能体的编排**：使用大语言模型将推断出的规则优化并编排为规则图（第 4.3节）。最终，为 $\mathcal{M}$ 中每个成功处理的三元组生成一个迁移程序。

4.1 `PolyglotPiranha`

我们在 `PolyglotPiranha` 中推断迁移脚本。 `PolyglotPiranha` 是 Uber [25] 开发并部署的一种用于源到源转换的领域特定语言。 `PolyglotPiranha` 体现了日益兴起的一类现代变换 DSL [4, 9, 45]，旨在支持工业规模的自动化重构。这些工具提供了一种结构化、声明式的替代方案，以取代临时编写的迁移脚本。

为进一步阐述第 2 节介绍的关键语言构造： `PolyglotPiranha` 程序由局部语法重写规则构成，这些规则组织成一个有向图，用以编码应用顺序、作用域约束和规则间依赖关系。每个节点定义一条规则，其中包含一个匹配模式和一个可选的重写子句；边使用作用域标签表示控制流。也就是说，一条边 $\mathcal{R}_1 \xrightarrow{\text{函数}} \mathcal{R}_2$表示：“应用规则 $\mathcal{R}_1$，然后在包含 $\mathcal{R}_2$ 的函数中应用规则 $\mathcal{R}_1$”。规则本身以具体语法模式编写（与目标语言的表层语法十分相似）。这些具体模式由字符串和抽象构成，可直接匹配具体代码。这些抽象通过模板变量表示（例如 `:[x]`），它们会绑定到具体的子表达式。例如，规则 `foo(:[args+])` $\mapsto$ `bar(:[args])` 将`foo(1,2,3)` 重写为 `bar(1,2,3)`，其中 `:[args]` 在转换过程中绑定到 `(1,2,3)`。

`PolyglotPiranha`程序以深度优先遍历规则图的方式运行。运行时从种子全局规则队列开始，并以深度优先方式应用这些规则，直到达到不动点。对于完整的语法与语义，我们建议感兴趣的读者参阅相关材料和文档 [19, 25]。

因此，对于每个迁移三元组（s、t、m） $\in$M，我们旨在综合一个 `PolyglotPiranha` 脚本 P = (R、E)，其中：

- R $=\{r_1$、r_2，... $r_k\}$ 是一组转换规则
- E $\subseteq$R $\times$L $\times$R 是一组有向边，其标签来自 L $=\{$文件, 函数,类 ...$\}$

每条规则 $r_i \in \mathrm{R}$ 都是一个元组 (p_i, q_i)，其中 p_i是匹配模式，q_i是替换模式。有向边 $(r_i, \ell, r_j) \in \mathrm{E}$ 表示应用规则 r_i 后，应在作用范围 ℓ 内应用规则 r_j。

4.2 原子规则推断

令 $\Delta(s, m)$ 表示源实现 s 与迁移 m之间的差异代码块集合。对于每个差异代码块 $h \in \Delta(s, m)$，令 h^- 和 h^+ 分别表示删除行和新增行。

我们使用反统一算法将这些代码块抽象为可复用的重写规则。反统一会计算两个表达式的最具体泛化，生成一种模式，该模式能够捕捉它们的共同结构，同时通过占位符抽象差异。这使其非常适合用于合成迁移规则，既能保留底层结构，又允许进行泛化。我们改编了一种现有的反统一算法 $\mathcal{A}$（来自Melt [39]），以推断初始规则集：

$$\mathcal{R}_0 = \{r : r = \mathcal{A}(h^-, h^+) \text{ for } h \in \Delta(s, m) \text{ where } h^- \neq \emptyset\}.$$

每条规则都会抽象前置和后置代码块中的共有元素。

例如，给定以下来自较大规模迁移的代码块： `cryptography` 到 `py-cryptodome`

```
- encrypted_data = fernet.encrypt(data)
+ padded_data = pad(data, AES.block_size)
+ encrypted_data = iv + fernet.encrypt(padded_data)
```

反统一（包括用于避免过度泛化的启发式方法）会生成：`:[x1]= :[x2].encrypt(:[x3])`

```
↦ padded_data = pad(:[x3], AES.block_size)
  :[x1]= iv + :[x2].encrypt(padded_data)
```

请注意，我们改编的算法不支持为仅添加而不删除代码的代码块生成规则 [39]；我们将在第6节讨论这一局限性。

4.3 脚本合成

规则推断提供初始规则集 R_0，，但不提供组合策略。智能体脚本合成通过迭代优化构建完整脚本 $\mathrm{P} = (\mathrm{R}, \mathrm{E})$。这一优化过程还允许 Spell 修正初始规则 r，因为这些规则可能存在过度泛化或错误。软件任务的智能体方法使用大语言模型迭代地采取行动并观察其影响，从而优化用于分解和解决复杂问题的策略。

图4 展示了工作流程。它以迁移三元组$(s, t, m) \in \mathcal{M}$ 以及初始规则集$\mathcal{R}_0$ 作为输入，该规则集是根据源实现和目标实现之间的差异 $\Delta(s, m)$ 推断得到的。模型的任务是制定包含完整脚本的总体策略，同时消解冲突规则、确定作用域和应用顺序，并在必要时改进转换规则，例如解决命名问题。本节其余部分将详细介绍该方法的关键组件。

系统提示词。 我们假设模型不熟悉相对较新的 `PolyglotPiranha`语言。系统提示词通过以下内容解释语言语法、语义和运行时行为：(1) 简单规则示例、具体语法说明，以及匹配/重写指南；(2) 对规则图、规则应用策略、作用域，以及用于在规则之间传播模板变量的构造的说明；(3) 迁移前后的源代码示例，并详细解释 `PolyglotPiranha` 如何执行迁移和运行时跟踪；最后是 (4) 其他指南、常见陷阱和最佳实践。

智能体循环。 对于每个迁移三元组（s、 t、 m）， $\in$M 和初始规则集，Spell 的智能体循环会反复采取行动并观察其效果，直到生成一个 `PolyglotPiranha` 程序，将源代码 s_i 转换为能够通过原始测试套件 t_i的迁移 $\hat{m}_i$，或直到完成 10 次迭代。

第一次迭代会使用一个部分完成的任务模板，其中包含迁移三元组和由 Spell 初步推断出的规则。我们提示模型生成一个将 s_i 转换为 m_i 的 `PolyglotPiranha` 程序，并遵循系统指南。提示中说明，生成的程序无需与 m_i 实现精确的令牌级匹配，但应体现预期的语义转换。我们（以及智能体）通过运行最终程序生成的代码来验证其正确性，测试套件为 t_i。

在每次迭代中，模型都会选择一个操作，并在受控环境中执行，执行结果将作为反馈返回：

- **优化 / 创建原子规则**。 模型可以创建一条新的匹配替换规则，也可以优化现有规则，并将其单独应用于 s_i，以观察该原子规则独立于其他转换时的行为。
- **向图中添加规则**。 模型可以向（最初为空的）规则图中添加一条或多条新规则。此操作会指定规则顺序和作用范围。
- **修订当前规则图**。 模型可以通过修改图的部分内容或重新生成图来修订规则图，而不局限于优化原子规则，从而实现对转换策略的高层次重构。
- **测试迁移**。 模型可以使用测试 t_i，将当前规则图应用于 s_i，并针对结果运行测试 t_i，以验证当前规则图。如果测试通过，循环便会终止。

环境。 Spell 在结构化环境中执行操作，并将结果反馈给模型，以引导下一步智能体操作。对于规则细化与集成以及规则图修订，Spell 使用 `PolyglotPiranha` 解释器将生成的 P 应用于 s_i，并将迁移后的程序作为反馈返回。如果有其他反馈，Spell 也会一并提供。例如，如果应用 P 因语法错误或 `PolyglotPiranha` 运行时错误而失败，Spell 会返回错误消息，并附上可能原因及修正建议。这些内容源自语言引擎中的常见故障模式和已知陷阱，类似于编译器诊断消息。如果脚本运行了，但没有修改 s_i，Spell 会根据我们使用该工具的经验，提供可能的解释和常见陷阱。对于测试，Spell 会在容器化测试环境中应用脚本并运行转换后的代码。如果所有测试均通过，Spell 会将此次迁移记录为成功；否则，会返回测试失败信息和错误消息，以供下一轮迭代使用。

有了迁移脚本后，我们就可以用它们将任何代码库从源库更新到目标库，因为这些规则与项目无关。负责迁移的开发者可以查看这些简洁的规则，了解所需的变更，然后使用 `PolyglotPiranha` 自动应用这些规则。

5 评估

我们将回答四个研究问题：

研究问题 1。（端到端有效性）：Spell生成 `PolyglotPiranha`迁移脚本的效果如何？我们评估整个流程生成可用`PolyglotPiranha` 脚本的程度。**研究问题 2。（现实适用性）：Spell的脚本在迁移现实世界中的客户端代码时有多大用处？**我们测试基于合成示例训练的 Spell 生成的 `PolyglotPiranha` 脚本能否成功迁移真实的开源项目，以评估其实用性。

研究问题 3。（与先前工作的比较）：Spell 与此前用于自动化重构的工具 Melt 相比表现如何？我们将 Spell 与 Melt进行比较。Melt 是此前最具可比性的 Python 迁移规则推断方法。

研究问题4。（数据质量）：数据生成流水线生成的合成迁移示例质量如何？Spell 的一项核心创新是通过合成示例进行自举。我们评估这些示例的有效性和多样性，并考察它们对源 API、目标 API 及其实现的覆盖是否充分。

5.1 实验设置

实现。我们在Spell 中 Python实现了相关功能。合成数据生成和智能体合成均与模型无关，并使用 OpenAI Chat Completion API [36] （大多数主要大语言模型 API 提供商均采用该 API）。在数据生成方面，我们使用高性价比的 `gpt-4o-mini` （小型模型），依据先前研究的发现：增加生成尝试次数可以弥补模型规模较小的不足 [44]。在合成方面，我们使用 `gpt-4.1` [35]，这是一款当前最先进的模型，具备更强的能力和更大的上下文窗口（100 万个 token），能够满足我们对大型系统提示词和链式交互的需求。

迁移任务。我们在 10 个迁移任务上进行评估，这些任务涉及 20 Python 个库；表1 的左侧对此进行了总结。这些任务经过挑选，涵盖了多个领域中各种广受欢迎的库及其替代方案，这些也是以往研究的目标 [14, 21, 34, 39]。我们重点关注 Python ，因为尽管 Python [30]是最受欢迎的编程语言之一 [6]，但针对它的重构工具却明显缺乏；这也使其成为基于大语言模型的工具的良好目标 [24]。

设置。对于每个迁移任务 (S, T)，我们生成 100 个用例。我们为每个用例 $C_j \in \mathrm{C}$ 生成 5 个源实现 $\tilde{s}_{j,k}$ ；为每个源实现 $\tilde{s}_{j,k}$生成 5 个测试文件 $\tilde{t}_{j,k,\ell}$ ；并为每个源实现 $\tilde{s}_{j,k}$生成 5 次迁移尝试 $\tilde{m}_{j,k,\ell}$ 。因此，每个任务最初包含 500 个实现、2,500 个测试文件和 2,500 次迁移尝试；随后按照第 3.2 节所述对其进行验证和筛选。我们对每个已验证的三元组运行合成流水线，以尝试生成脚本 $\mathrm{P} = (\mathrm{R}, \mathrm{E})$。我们允许智能体最多迭代 10 次来生成脚本。

我们选择这些数值（10、5）以平衡成本与覆盖率：根据模型定价，这一配置让我们能够将总预算控制在 100 美元以内，同时仍能得出有意义的结果。由于迁移问题更难，我们生成了更多迁移尝试；多次尝试可以提高成功的概率 [44]。

如果脚本替换了所有源库 API 调用（$\mathrm{APIs}(\mathrm{P}(s), \mathrm{S}) = \emptyset$），并且将其应用于 s_j 后，迁移后的代码通过了与 $m_{j,k}(\mathrm{P}(s) \equiv_t m)$ 相同的测试，则认为综合尝试成功 。

5.2 RQ1：端到端有效性

方法。 我们首先端到端评估合成流水线。对于每项迁移任务，我们报告在所有用例中生成的有效三元组数量（表1第 5 列“有效三元组”）；这是数据生成的输出（第3节）。成功率（Spell，表1 第 6 列）表示脚本合成生成成功 PolyglotPiranha程序的有效示例所占百分比。我们在第5.5节研究合成数据的质量和多样性。为评估脚本泛化能力，我们将每个 $\mathrm{P}_{j,\,k}$（来自三元组（$s_{j,\,k}$，$t_{j,\,k,\,\ell}$，$m_{j,\,k,\,r}$））应用于同一用例 C_j 的同级实现 $\{s_{j,\,i} : i \neq k\}$。我们依据测试报告生成的脚本成功迁移了多少个同级实现。

表 1. 迁移任务及其合成结果概览。有效三元组：生成的迁移三元组中覆盖率为>60%。成功率：各工具成功生成经验证的 PolyglotPiranha 脚本的迁移三元组百分比。同级实现成功率：至少有一个合成的PolyglotPiranha 脚本成功迁移的同级实现数量，占潜在可迁移同级实现总数的比例。

迁移任务				有效	成功率（%）		兄弟组件
源语言	目标语言	领域	星级（源语言 / 目标语言）	三元组	Spell	MELT	成功
argparse	click	CLI	不适用 / 16.4k	215	44.2	17.2	91/170
jinja2	mako	模板引擎	10.9k / 389	15	13.3	0.0	0/3
json	orjson	JSON 序列化	不适用 / 6.9k	269	96.7	57.6	258/291
logging	loguru	日志记录	不适用 / 21.7k	114	85.1	72.8	64/93
lxml	BeautifulSoup	HTML/XML 解析	2.8k / 不适用	32	59.4	0.0	0/21
pandas	polars	数据分析	45.5k / 33.8k	0	-	-	-
pathlib	pyfilesystem2	文件系统抽象	不适用 / 2.0k	21	52.4	28.6	1/15
cryptography	pycryptodome	cryptography	7.0k / 3.0k	79	48.1	6.3	3/46
requests	httpx	HTTP 客户端（异步、同步）	52.9k / 14.1k	65	76.9	12.3	39/63
time	pendulum	日期/时间	不适用 / 6.4k	60	78.3	11.7	35/73

结果。 表 1 展示了结果。 Spell 为十项任务中的九项成功生成了有效迁移三元组，每项任务平均生成 87 个经过筛选的有效三元组（ pandas → polars 例外情况将在第 5.5节讨论）。使用这些已验证示例，Spell的迁移合成脚本平均成功率为 61.6% ，每次仅进行单次试验。通过测试时扩展，这一比率可能会进一步提高 [44]；考虑到成本，我们将其留待未来工作。

表格 1 的最后一列显示，迁移脚本成功迁移了 63.3% 的同级实现（491/774，见表格1的最后一列）。对于简单的一对一 API 替换，效果尤其出色，例如 json → orjson （88.6%）；而涉及嵌入式领域专用语言的迁移（jinja2 →mako， lxml → beautifulsoup）泛化效果则较差。

这些库支持动态网页生成，并在Python 字符串字面量中嵌入模板语言，而 PolyglotPiranha 引擎无法转换这些内容；这是 PolyglotPiranha 语言本身的局限 [25]。尽管存在这一限制，我们合成的规则仍能有效捕捉大多数迁移任务中的常见使用模式。

5.3 RQ2：现实世界适用性

方法。 为了评估我们推断出的迁移脚本能否推广到合成示例之外，我们将其应用于托管在 GitHub 上的 18 个 Python 项目。我们使用 Sourcegraph [1]收集了一个便利样本，其中包含调用了生成脚本中所涉及源 API 的代码库，并按星标数排序，优先选择包含测试套件的代码库。由于让每个项目成功构建并通过测试需要投入大量工程工作量，我们仅针对十项迁移任务中的六项开展了此项评估。

我们手动审查了每个项目，以找出与观察到的 API 使用模式最匹配、最相关的生成式迁移脚本。我们根据每个项目的文档化设置，使用基于 Dockerfiles 的容器化环境执行代码，运行测试以确定基线功能，然后应用选定的 PolyglotPiranha 脚本并重新运行测试。我们更新了 requirements.txt 以包含目标依赖项，但保留了源依赖项，因为我们的脚本通常无法涵盖库中的所有 API 用法。由于 PolyglotPiranha 的引擎几乎能即时执行，我们没有测量转换所需的 time。这些复杂的代码库通常包含数百个文件和数千行代码。

表 2. 应用于真实世界代码仓库的推断迁移脚本。每一行代表将迁移脚本应用于使用源库的代码仓库。**星标** 和 **千行代码** 分别表示 GitHub 热度和代码库规模。**重写次数** 是某条 `PolyglotPiranha` 规则在项目中触发的次数。通过的测试（前/后/%）：迁移前后通过的测试用例数，以及测试保留率（迁移后/迁移前），用于表明基本功能得以维持。

迁移任务	代码仓库	星星	KLoC	通过的测试			
				重写	之前	之后	%
`json → orjson`	`BIM2SIM/bim2sim`	57	33.8	46	173	149	86%
	`kaapana/kaapana`	196	69.8	317	45	43	96%
	`SoCo/SoCo`	1500	17.4	7	233	233	100%
`logging → loguru`	`fastqe/fastqe`	172	0.7	16	2	2	100%
	`mie-lab/trackintel`	229	11.2	14	411	411	100%
	`pywbem/pywbem`	42	171.7	16	2499	0	0%
`lxml → BeautifulSoup`	`acl-org/acl-anthology`	528	19.4	111	433	433	100%
	`pyxnat/pyxnat`	73	7.7	8	62	62	100%
	`w3af/w3af`	43	201.8	10	2547	2547	100%
`cryptography → pycryptodome`	`aomail-ai/aomail-app`	134	2.0	1	13	11	85%
	`BM/qpylib`	32	2.0	4	139	127	91%
	`DMcP89/harambot`	32	1.6	8	8	0	0%
`requests → httpx`	`Kildrese/scholarBibTex`	33	0.1	5	1	1	100%
	`wjohnson/pyapacheatlas`	175	13.9	22	128	128	100%
	`databricks/databricks-sdk-py`	433	108.6	23	157	157	100%
`time → pendulum`	`clld/clld`	57	9.1	4	308	308	100%
	`fbpic/fbpic`	192	25.5	43	6	5	83%
	`qutip/qutip`	1800	41.2	29	1585	147	9%

结果。表2 报告了每个项目的重写次数（即某条 `PolyglotPiranha`规则被触发以转换代码的次数），以及迁移前后的通过测试数量。通过检查现有测试是否仍然通过，我们验证了维护者认为重要的功能得以保留。尽管脚本是根据合成示例推断得出的，但在许多情况下，迁移后所有测试都通过了。例如，对于 `logging → loguru`迁移，推断出的脚本在 `trackintel` 和 `fastqe` 项目中都保留了测试行为，并触发了大量重写。类似地，Spell为 `json → orjson` 和`requests → httpx` 编写的脚本在真实项目中应用了数十到数百处编辑，同时保留了功能行为。经人工检查，我们发现测试失败通常是由于迁移不完整：脚本正确处理了部分迁移，但仍需进一步完善或添加转换。

5.4 研究问题 3：与先前工作的比较

方法。 我们将自己的合成方法与 Melt [39], 这一最先进的自动化 `Python` 重构工具进行比较。Melt 是少数专为 Python 设计的工具之一，并基于转换领域专用语言构建 `comby` [47, 48]（参见第 7 节）。虽然 Spell 使用生成的或合成的示例，Melt 则从开发者拉取请求中挖掘的真实迁移对推断迁移脚本。为了公平比较（我们的迁移并非都与此类拉取请求相关），并更有效地单独考察 Spell 中的混合合成，我们在生成的迁移示例上运行 Melt。

结果。 表 1的第 6 和第 7 列比较了这两种方法。对于能够获得有效三元组的所有任务，Spell 的表现都优于 Melt，且通常优势显著。其平均成功率达到 61.6%，而 Melt为 22.9%，尽管这两种工具使用的起始数据都是相同的已验证示例。

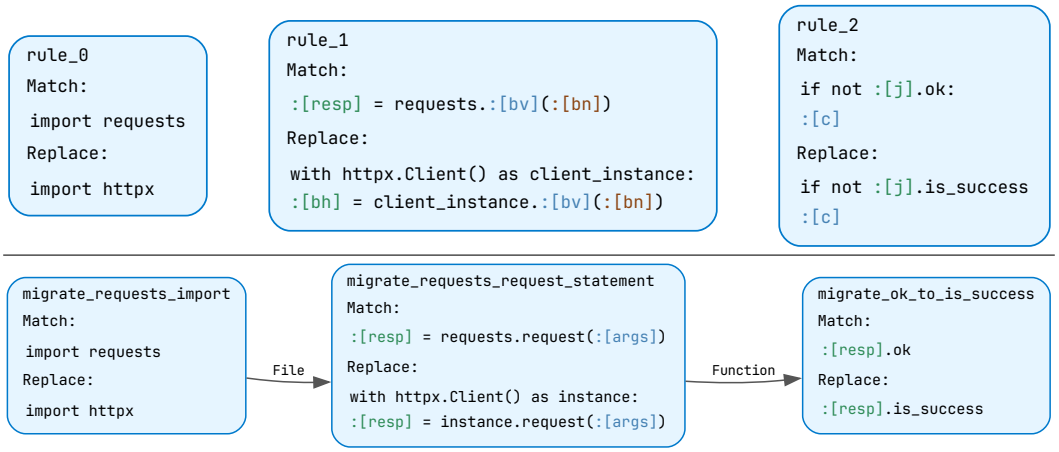

图5。Melt（上）和 Spell（下）根据 `requests` →`httpx`的一个迁移示例生成的程序。Melt 的规则包含通用的占位符名称，且过度抽象；Spell 的脚本包含作用域应用和具有语义意义的名称。

表3。 生成的迁移示例的质量：用例成功率、API 多样性和测试覆盖率 e

迁移任务	用例（共100个）		不同的 API		平均覆盖率		已验证
	已实现	迁移源代码		目标源代码		目标示例	
argparse → click	74	72	11	16	0.89	0.93	215
jinja2 → mako	24	14	9	4	0.86	0.86	15
json → orjson	96	96	4	3	0.70	0.70	269
logging → loguru	58	52	12	6	0.79	0.79	114
lxml → beautifulsoup	59	23	11	9	0.75	0.75	32
pandas → polars	58	2	5	4	0.43	0.43	0
pathlib → pyfilesystem2	64	23	10	17	0.71	0.70	21
cryptography → pycryptodome	72	52	59	25	0.73	0.72	79
requests → httpx	64	60	8	10	0.66	0.64	65
time → pendulum	81	55	9	34	0.61	0.61	60
平均值	**65.0**	**44.9**	**13.8**	**12.8**	**0.71**	**0.71**	**87.0**

事实上，Melt 能够从许多合成示例中提取转换规则，这进一步证明了这些示例的质量，也表明数据生成流水线或许能提升其他迁移推断工具的表现。然而，Melt的规则彼此孤立。相比之下，Spell的智能体流水线会将这类规则编排成连贯的转换策略，从而有效利用 `PolyglotPiranha` 语言的表达能力。图 5 展示了这种差异：Spell合成相互关联的规则，并明确限定作用域（按函数、按文件），以实现级联转换，同时避免过度泛化；Melt则生成互不关联的全局规则，并使用通用占位符，这在大型代码库中尤其容易造成问题 [25]。

这些结果验证了：（1）生成的合成示例为迁移推断提供了有用的独立信号；（2）用于规则组合的智能体方法显著优于当前最先进水平，而对推断规则进行智能编排也为迁移合成做出了重要贡献。

5.5 研究问题4：数据质量

方法论 我们从多个维度评估合成生成数据的质量：

（1）用例实现：100 个生成的用例 $C_j \in \mathrm{C}$ 中，至少产生一个实现$s_{j,k}$ 且通过至少一个生成的测试框架的数量。（2）用例迁移：已实现用例中迁移到目标库的数量（有多少用例至少有一个 s_i 与一次成功的 m_i相关联）。（3）API 多样性：来自 S 和 T 的不同 API 方法数量。[3] 我们使用 Jedi 静态分析器 [18] 在已验证三元组 $(s, t, m) \in \mathrm{M}$ 上计算该指标。（4）测试覆盖率：通过生成的测试在源实现和目标实现上的行级覆盖率（根据我们对有效性的定义，覆盖率必须至少为 0.6）。（5）已验证示例：所有生成示例在过滤后的已验证示例总数；这决定了可用于脚本合成的训练数据。

这些指标共同刻画了所生成合成数据的质量，具体体现在其对迁移任务的覆盖或表示程度，以及测试是否充分。

结果。 表 3 展示了所有数据质量指标的结果。 Spell成功实现了大多数生成的用例，平均每 100 个用例中有 65 个至少产生了一个可运行的实现。成功率介于 24（`jinja2 → mako`）和 96（`json → orjson`）之间。通过人工检查 `jinja2 → mako`，我们发现，该任务的大多数测试代码都使用字符串比较来验证预期输出，而模型难以正确处理这些比较。由于不同库使用不同的字符串表示来表示模板中的占位符，这些测试也无法在库之间迁移。

在至少实现过一次的用例集合中，Spell 根据生成的测试，在保持功能等价性的同时，平均成功迁移了 44.9 个用例。异常值是 `pandas → polars`，尽管成功实现了 58 个用例，但仅成功迁移了 2 个。人工检查显示，生成的测试框架依赖于 pandas 特有的构造（例如，`DataFrames`），而这些构造与 `polars` 不兼容。

我们使用 Jedi 对 API 多样性的分析发现，每个迁移任务平均涉及 13.8 个不同的源库 API 和 12.8 个目标库 API。这些 API 可能是底层大语言模型在预训练期间最常遇到的 API [24]。 `cryptography →pycryptodome` 任务表现出最高的多样性（源库 59 个、目标库 25 个 API），而 json → orjson 这类简单任务使用的 API 较少（源库 4 个、目标库 3 个）。

生成的测试对源代码和目标实现的平均覆盖率均达到 71%，远高于 60% 的过滤阈值。覆盖率介于 61%（`time → pendulum`）和 89%/93%（`argparse → click`）之间。这表明测试能够有效检验实现逻辑，但仍有提升空间；众所周知，大语言模型在单元测试生成方面表现不佳 [40]。

应用所有过滤标准后，Spell 为每个任务平均生成了 87 个已验证的迁移三元组（不包括 `pandas → polars`）。总体而言，这些示例可测试且具有多样性（每个库平均涵盖 13–14 个不同的 API），同时对源代码和迁移后的代码也都实现了较高的覆盖率（平均 71%）。这表明它们可用作下游合成的输入。

6 讨论与局限性

合成数据的质量与代表性。 我们的流程依靠由规模相对较小的模型（`gpt4o-mini`）生成的合成示例，以便将数据生成的成本控制在可承受范围内。我们证明，这对于我们的方法已经足够，而且生成的数据也可供

[3]由于 `Python` 采用动态类型，因此应将此启发式方法视为下限。

先前的一种迁移推断方法使用。不过，大语言模型生成的程序通常偏向于各个 API 中使用频率特别高的部分。边界情况的行为、错误处理路径以及很少使用的配置参数可能代表不足。随着使用更多数据训练的更大型模型变得更加便宜，我们预计 API使用的广度以及所生成程序的正确性和质量都会提升。另一个互补方向是在用例—实现提示中加入对真实代码仓库的检索。这对于某种用例尤其有帮助：该用例使客户端能够描述相关的 API 方法，从而指导样本生成，并可能提高迁移规则在真实世界代码仓库上的覆盖率。

适用性。 我们使用 `PolyglotPiranha` 语言表示迁移，这是因为它具有表达力强且简洁的规则图抽象，同时性能出色。然而， `PolyglotPiranha`仍有局限，尤其是尚不能重写字符串字面量、格式化模板或动态生成结构中的代码。这一局限是 `jinja2→mako` 任务成功率较低的主要原因。在 `PolyglotPiranha`中集成多语言解析，或改用支持嵌入式领域专用语言的混合转换引擎，都可能提升此任务的性能。

第 5 节表明，这项技术在某些迁移任务中前景可观（例如，代码操作标准数据类型，或生成明确定义的输出，以便跨库轻松比较），但在其他任务中效果不佳。在实际应用中，一个主要局限是开源项目中的代码存在诸多变体。 Melt 推断出的原子规则能够忠实地捕捉具体的差异代码块，但可能过于具体，无法匹配细微的语法变体；也可能过于宽泛，从而引入误报。 Spell的智能体循环通过优化和测试规则缓解了其中一些问题，但其本质上仍受示例限制：如果某种使用模式未出现在任何迁移三元组中，生成的脚本就无法处理它。 `Pycraft` [13]关于使用大语言模型进行规则泛化的技术，可以集成到我们的工作流中，生成更多等效的脚本变体，以扩大适用范围。未来的工作还可以通过纳入项目特定的 API 使用模式来优化迁移脚本，并利用失败测试的反馈修复剩余问题，从而完成部分迁移。

测试。 Spell的验证流程的核心支柱之一是使用自动生成的测试来评估实现及其迁移后对应版本的正确性。虽然这使得无需人工操作即可进行大规模验证，但也带来了一些局限性。首先，测试的质量在很大程度上取决于生成模型生成稳健且有意义的测试框架的能力。尽管我们通过为每个实现采样多个测试文件，并应用基于覆盖率的筛选来缓解这一问题（如第 3.1.3 节所述），测试仍可能较为浅显，或无法检验边界情况行为。特别是，测试只能确认基本的功能等价性（例如输出匹配），因此可能会遗漏细微的行为差异或副作用。其次，我们的方法假设通过相同测试套件足以作为语义等价性的代理指标。尽管这一假设对许多情况而言是合理的，但在形式上并无保证。测试套件可能缺乏完整性或针对性。第三，我们基于覆盖率的筛选有助于确保测试执行实现逻辑中非平凡的一部分，但并不能保证语义健全性。例如，执行了大部分代码但断言较弱的测试（例如仅断言未引发异常）即使行为不正确，也可能通过。其他验证技术（例如变异测试 [23]）可能有助于缓解这一问题。

7 相关工作

代码转换工具集。代码转换工具集各不相同。用于抽象语法树重写的命令式框架包括 Java 的 `errorprone`[2] 和 `OpenRewrite`[9]。 `clang libtooling`[8]

通过 AST匹配器提供声明式匹配，同时提供用于重写代码的命令式框架。

此外，也存在纯声明式的代码变换工具，例如 `coccinelle` [32]、`GritQL` [45]、 `ast-grep` [4]、`comby` [48]以及 `PolyglotPiranha` [25]。 `comby` 是 `PolyglotPiranha`的思想先驱，我们在本文中使用后者。 `PolyglotPiranha` 基于 `comby`模型构建，通过级联转换（边）、显式作用域和过滤器，允许组合规则以实现细粒度且限定作用范围的重写。我们选择用`PolyglotPiranha` 表示迁移脚本，因为它已在工业规模的应用中取得显著成功， [25] 而且语法简单，适合用于合成。

基于示例的转换。 许多工具会从示例中推断迁移规则 [3, 14, 16, 17, 26,31, 33, 51]。这些技术大多假定能够获取从一个库手动迁移到另一个库的项目，但这样的示例很少见 [46]。要找到这些示例，通常需要挖掘大型代码库并重构变更历史 [20]。Spell 可以补充这些方法，自动从大语言模型中提炼代码迁移知识，并生成可作为此类工具训练数据的示例，从而减少对人工整理的迁移历史的依赖。

直到最近，这些工具才开始利用 Melt [39]、 `PyEvolve` [14] 和 `TCInfer` [26, 43] 等可用于生产环境的转换引擎，它们使用 `comby` 语言表达迁移逻辑。其中，Melt 与我们的工作最为相关：它针对破坏性变更，而 Spell 专注于跨库迁移。我们使用 Melt 的推断算法生成初始规则集，然后采用智能体方法对其进行优化，从而生成完整的 `PolyglotPiranha` 迁移脚本。我们的方法也可以受益于近期有关转换规则泛化的研究，尤其是 `Pycraft` [13]；该方法为单条规则生成多个变体，以扩大适用范围，解决了 `PolyglotPiranha` 和 `comby` 都存在的一个已知局限（即如何将规则泛化到推断它们所依据的特定示例之外）。

用于代码变换的大语言模型。近期的发展表明，人们越来越关注使用大语言模型自动化代码迁移。 `Amazon Q Developer`[42] 等业界工具以及有关借助大语言模型进行 Java 升级的报告[52] 都反映了这一趋势。学术界的相关工作包括 `DeepMig`[11] 和 Islam等人[21]。`DeepMig` 依赖基于迁移对的监督训练，而 Spell 无需挖掘语料库，便能提取自己的示例。

`SOAR` [34] 是最早尝试利用语言模型进行迁移的方法之一。它使用 API文档推断源库与目标库中函数之间的对应关系，然后通过程序合成迁移客户端代码。与 Spell 不同，它不会生成迁移示例或转换脚本。这两种方法都利用语言模型获取迁移知识：Spell 通过提示和预训练， `SOAR` 则通过嵌入相似度。Melt [39] 利用引入破坏性变更的拉取请求，提示大语言模型生成从旧版本到新版本的过渡示例，并将模型知识与示例代码和测试相结合。Spell 使用更为复杂的自指令循环来生成用例、实现和迁移，将 Melt 的方法扩展到更广泛的领域。

8 结论

库迁移是开发者经常需要承担且劳动密集的任务。我们提出了 Spell，这是一种从大语言模型中提取迁移知识，并将其转换为可复用`PolyglotPiranha` 脚本的系统。Spell 不依赖手动迁移的语料库，而是提示 LLM 为源库与目标库生成代码和测试，然后利用反统一与智能体合成提炼出通用转换。生成的脚本兼容现代重构工作流，可供检查、版本化，并应用于大型代码库。

我们在十项 Python迁移任务上评估了 Spell，生成了 870 个已验证的迁移示例。每个示例仅进行一次综合尝试时，Spell 为 61.6% 的案例生成了正确脚本，并迁移了 63.3% 的同级实现，始终优于 MELT。在 18 个真实世界的开源仓库上应用时，Spell 完成了数百项编辑，并在大多数案例中保持了测试行为。这些结果表明，LLM 编码了可操作的迁移知识，这些知识能够被提取、验证，并形式化为易于维护的转换逻辑。

参考文献

[1] [n. d.]. Sourcegraph。https://sourcegraph.com/。访问日期：2025年7月18日。[2] Edward Aftandilian、Raluca Sauciuc、Siddharth Priya 和 Sundaresan Krishnan。2012。使用可扩展的 Java 编译器构建实用的程序分析工具。IEEE Computer Society，意大利 Riva del Garda，

14 23 10 1109 2012 28 – 。

[3] doi：。/SCAM. 。Jesper Andersen 和 2008 Julia L. Lawall。 。通用补丁推断。发表于自动化软件工程国际会议（ASE）论文集。IEEE Computer Society，

337 346 10 1109 2008 44 – 。doi：。/ASE. 。Astgrep。 。

[4] 2024 ast-grep：编写代码来匹配代码。开源。https://ast-grep.github.io 访问日期：2024-03-06。[5] Christopher Bogart、Christian Kästner、James D. Herbsleb 和 Ferdian Thung。2016。如何破坏一个 API：三个软件生态系统中的成本协商与社区价值观。发表于 ACM SIGSOFT软件工程基础研讨会（FSE）论文集，Thomas Zimmermann、Jane Cleland-Huang 和 Zhendong Su（编）。ACM，109–120。[6] TIOBE Software BV。2025。TIOBE指数。https://www.tiobe.com/tiobe-index/。访问日期：2025年5月5日。[7] Ned Chapin、Joanne E Hale、Khaled Md Khan、Juan F Ramil 和 Wui-Gee Tan。2001。软件演化与软件维护的类型。《软件维护与演化：研究与实践》13，1（2001），3–30。[8] Clang。2024。Clang LibTooling。访问日期：2024-03-06。[9] OpenRewrite贡献者。2025。OpenRewrite：让重构更安全。https://docs.openrewrite.org/。访问日期：2025-03-27。[10] Cleidson RB De Souza、David Redmiles、Li-Te Cheng、David Millen 和 John Patterson。2004。一种良好的软件实践如何阻碍协作：API在软件开发中的多重作用。《ACM SIGSOFT软件工程简报》29，6（2004），221–230。[11] Juri Di Rocco、Phuong T Nguyen、Claudio Di Sipio、Riccardo Rubei、Davide Di Ruscio 和 Massimiliano Di Penta。2025。DeepMig：一种支持耦合库与代码迁移的基于Transformer的方法。《信息与软件技术》177（2025），107588。[12] Danny Dig 和 Ralph E. Johnson。2006。API如何演化？重构的故事。《软件维护：研究与实践》18，2（2006），83–107。[13] Malinda Dilhara、Abhiram Bellur、Timofey Bryksin 和 Danny Dig。2024。前所未有的代码变更自动化：大语言模型与基于示例的转换的融合。《ACM软件工程论文集》1，FSE（2024），631–653。[14] Malinda Dilhara、Danny Dig 和 Ameya Ketkar。2023。PyEvolve：自动化Python机器学习系统中的频繁代码变更。发表于 ACM IEEE国际软件工程会议（ICSE）论文集。[15] Mattia Fazzini、Qi Xin 和 Alessandro Orso。2019。Android应用的自动化API用法更新。发表于软件测试与分析国际研讨会论文集，Dongmei Zhang 和 Anders Møller（编）。ACM，204–215。[16] Mattia Fazzini、Qi Xin 和 Alessandro Orso。2020。APIMigrator：Android应用的API用法迁移工具。发表于 IEEE/ACM第七届移动软件工程与系统国际会议论文集，David Lo、Leonardo Mariani 和 Ali Mesbah（编）。77–80。[17] Xiang Gao、Arjun Radhakrishna、Gustavo Soares、Ridwan Shariffdeen、Sumit Gulwani 和 Abhik Roychoudhury。2021。APIFix：用于应对库中破坏性变更的面向输出的程序合成。发表于 AC M SIGPLAN面向对象编程、系统、语言与应用（OOPSLA）论文集，第5卷。1–27。[18] David Halter。 [n. d.]。Jedi：适用于Python的自动补全工具。https://jedi.readthedocs.io/en/latest/。访问日期：2025年4月25日。[19] Uber Technologies Inc. 2025。Piranha：用于重构与特性开关API相关代码的工具。https://github.com/ uber/piranha 访问日期：2025-04-10。[20] Mohayeminul Islam、Ajay Kumar Jha、Ildar Akhmetov 和 Sarah Nadi。2024。Python库迁移特征分析。《ACM软件工程论文集》1，FSE（2024），92–114。[21] Md Mohayeminul Islam、Ajay Kumar Jha、May Mahmoud、Ildar Akhmetov 和 Sarah Nadi。2025。使用大语言模型进行库迁移。arXiv预印本 arXiv:2504.13272（2025）。[22] Kevin Jesse、Toufique Ahmed、Premkumar T Devanbu 和 Emily Morgan。2023。大语言模型与简单而愚蠢的错误。发表于2023年IEEE/ACM第20届软件仓库挖掘国际会议（MSR）。IEEE，563–575。[23] Yue Jia 和 Mark Harman。2010。变异测试的发展分析与综述。《IEEE软件工程汇刊》37，5（2010），649–678。

[24] Juyong Jiang、Fan Wang、Jiasi Shen、Sungju Kim 和 Sunghun Kim。2024。代码生成领域大语言模型综述。arXiv预印本 arXiv:2406.00515（2024）。[25] Ameya Ketkar、Daniel Ramos、Lazaro Clapp、Raj Barik 和 Murali Krishna Ramanathan。2024。一种轻量级多语言代码变换语言。《ACM编程语言论文集》 8, 2024 1288 1312 10 1145 3656429 PLDI（），–。[26] doi：。/ Ameya Ketkar、Oleg Smirnov、Nikolaos Tsantalis、Danny Dig 和 Timofey Bryksin。 。推断并应用类型变更。发表于 ACM IEEE国际软件工程会议（ICSE）论文集。ACM，1206–1218。[27] Miryung Kim、Thomas Zimmermann、Robert DeLine 和 Andrew Begel。2018。软件团队中的数据科学家：现状与挑战。发表于 ACM IEEE国际软件工程会议（ICSE）论文集，Michel Chaudron、Ivica Crnkovic、Marsha Chechik 和 Mark Harman（编）。585。[28] Miryung Kim、Thomas Zimmermann 和 Nachiappan Nagappan。2012。重构挑战与收益的实地研究。发表于 ACM SIGSOFT软件工程基础研讨会（FSE）论文集。ACM，50。[29] Raula Gaikovina Kula、Daniel M. Germán、Ali Ouni、Takashi Ishio 和 Katsuro Inoue。2018。开发者会更新其库依赖吗？——安全公告对库迁移影响的实证研究。《经验软件工程》23，1（2018），384–417。[30] Maxime Lamothe、Yann-Gaël Guéhéneuc 和 Weiyi Shang。2021。API演化文献的系统综述。《ACM计算调查》54，8（2021），1–36。[31] Maxime Lamothe、Weiyi Shang 和 Tse-Hsun Peter Chen。2022。A3：利用代码示例辅助Android API迁移。《IEEE软件工程汇刊》（TSE）（2022），417–431。[32] Julia Lawall 和 Gilles Muller。2018。Coccinelle：Linux内核中自动化演化的十年历程。发表于USENIX年度技术会议，Haryadi S. Gunawi 和 Benjamin C. Reed（编）。USENIX协会，美国马萨诸塞州波士顿，601–614。https://www.usenix.org/conference/atc18/presentation/lawall[33] Na Meng、Miryung Kim 和 Kathryn S. McKinley。2013。LASE：通过示例学习定位并应用系统性编辑。发表于 ACM IEEE国际软件工程会议（ICSE）论文集，David Notkin、Betty H. C. Cheng 和 Klaus Pohl（编）。502–511。[34] Ansong Ni、Daniel Ramos、Aidan Z. H. Yang、Inês Lynce、Vasco M. Manquinho、Ruben Martins 和 Claire Le Goues。2021。SOAR：一种用于数据科学API重构的程序合成方法。发表于 ACM IEEE国际软件工程会议（ICSE）论文集。IEEE，112–124。[35] OpenAI。2025。GPT-4.1：在API中推出GPT-4.1。https://openai.com/index/gpt-4-1/[36] OpenAI。 [n. d.]。OpenAI Python API库。https://pypi.org/project/openai/。访问日期：2025年4月22日。[37] Jeff H. Perkins。2005。自动生成重构以支持API演化。发表于 ACM软件工具与工程程序分析研讨会论文集。111–114。[38] Daniel Ramos、Catarina Gamboa、Inês Lynce、Vasco Manquinho、Ruben Martins 和 Claire Le Goues。2025。Spell：使用大语言模型合成程序化编辑。https://figshare.com/s/08961f436816a434b77b。Figshare上的工件。[39] Daniel Ramos、Hailie Mitchell、Inês Lynce、Vasco Manquinho、Ruben Martins 和 Claire Le Goues。2023。MELT：从拉取请求中挖掘有效的轻量级转换。发表于自动化软件工程国际会议（ASE）论文集。IEEE, 1516 1528 10 1109 56229 2023 00117 –。doi：。/ASE。 。Nikitha Rao、[40] Kush Jain、Uri Alon、Claire Le Goues 和 Vincent J. Hellendoorn。2023。CAT-LM：基于对齐代码与测试训练语言模型。发表于第38届IEEE/ACM自动化软件工程国际会议，ASE *2023,* *11 15, 2023* 409 420 10 1109 56229 2023 00193 卢森堡，9月-。IEEE，–。[41] doi：。/ASE。 。Baptiste Rozière、Marie-Anne Lachaux、Lowik Chanussot 和 Guillaume Lample。 。编程语言的无监督翻译。发表于《神经信息处理系统进展》第33卷：2020年神经信息处理系统年会，NeurIPS 2020，2020年12月6日至12日，线上举行，Hugo Larochelle、Marc'Aurelio Ranzato、Raia Hadsell、Maria-Florina Balcan 和 Hsuan-Tien Lin（编）。https://proceedings.neurips.cc/paper/2020/hash/ed23fbf18c2cd35f8c7f8de44f85c08d-Abstract.html[42] 亚马逊云科技。2025。Amazon Q Developer：Transform。https://aws.amazon.com/q/developer/transform/。访问日期：2025-03-27。[43] Oleg Smirnov、Ameya Ketkar、Timofey Bryksin、Nikolaos Tsantalis 和 Danny Dig。2022。IntelliTC：在 IntelliJ IDEA 中自动化类型变更。发表于 ACM IEEE国际软件工程会议（ICSE）论文集。115–119。[44] Charlie Snell、Jaehoon Lee、Kelvin Xu 和 Aviral Kumar。2024。最优地扩展大语言模型的测试时计算，可能比扩展模型参数更有效。arXiv预印本 arXiv:2408.03314（2024）。[45] GritQL团队。2025。GritQL：代码转换查询语言。https://grit.io/gritql/。访问日期：2025-03-27。[46] Cédric Teyton、Jean-Rémy Falleri、Marc Palyart 和 Xavier Blanc。2014。Java中的库迁移研究。《软件演化与过程杂志》26，11（2014年11月），1030–1052。doi:10.1002/smr.1660[47] Rijnard van Tonder。2025。Comby。检索日期：2025年7月18日，网址：https://comby.dev/docs/overview[48] Rijnard van Tonder 和 Claire Le Goues。2019。使用解析器组合子进行轻量级多语言语法转换。发表于 ACM SIGPLAN编程语言设计与实现会议（PLDI）论文集。363–378。

[49] Yizhong Wang、Yeganeh Kordi、Swaroop Mishra、Alisa Liu、Noah A Smith、Daniel Khashabi 和 Hannaneh Hajishirzi。2022。Self-instruct：通过自生成指令对语言模型进行对齐。*arXiv* 预印本 *arXiv:2212.10560*（2022）。[50] Laerte Xavier、Aline Brito、André C. Hora 和 Marco Túlio Valente。2017。API 破坏性变更的历史与影响分析：一项大规模研究。载于*IEEE* 软件分析、演化与再工程国际会议论文集，Martin Pinzger、Gabriele Bavota 和 Andrian Marcus（编）。138–147。[51] Shengzhe Xu、Ziqi Dong 和 Na Meng。2019。Meditor：API 迁移编辑的推断与应用。载于程序理解国际会议论文集，Yann-Gaël Guéhéneuc、FoutseKhomh 和 Federica Sarro（编）。IEEE / ACM，335–346。[52] Celal Ziftci、Stoyan Nikolov、Anna Sjövall、Bo Kim、Daniele Codecasa 和 Max Kim。2025。在 Google 大规模迁移代码：使用 LLM。*arXiv* 预印本 *arXiv:2504.09691*（2025）。