

Agentic Dependency Management at Repository-Level

○ **Kaya Kitada** Yoshiki Higo Raula Gaikovina Kula

大阪大学 大学院情報科学研究科 コンピュータサイエンス専攻 ソフトウェア工学講座 M1

ソフトウェア工学研究会(SIGSE) 2026年7月



背景：現代のソフトウェアは外部ライブラリ依存の上に構築される

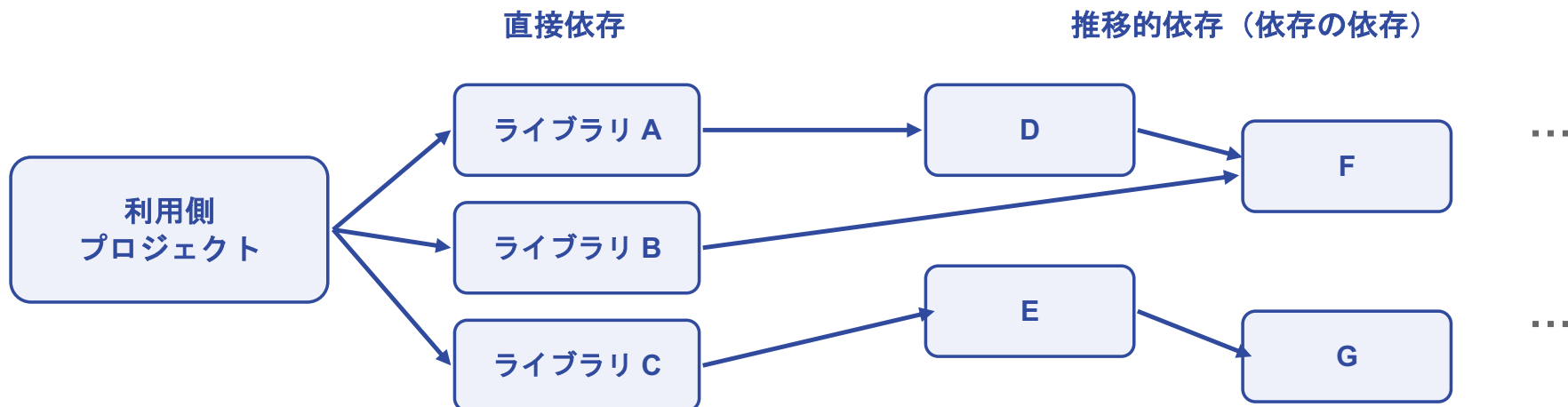
ライブラリ = 第三者が提供する再利用可能なモジュール

依存(関係) = 自プロジェクトの動作が外部ライブラリの機能に依存する関係

依存関係管理(NPM(JavaScript)・PyPI(Python)) = ライブラリエコシステム が存在

OSS がソフトウェアの基盤となっている

- 商用コードベースの 96% が OSS を含む
- コード全体の 77% が OSS 由来
- 出典：Synopsys, 2024 Open Source Security and Risk Analysis (OSSRA) Report



課題：依存の維持コストとライブラリ利用のリスク

更新の負担

ライブラリは頻繁にバージョンアップ。追従しないと動かなくなり、追従には手間がかかる

脆弱性

ライブラリのセキュリティ欠陥が、そのまま自プロジェクトの脆弱性になる

サプライチェーン攻撃

依存パッケージに悪意あるコードが混入すると、それを利用する全ソフトウェアへ波及

管理コストが便益を上回る状態 = **依存技術的負債**

多くのライブラリはボランティアが維持 → 更新停止・引き継ぎ問題のリスクも

→ こうした負債は根深く、手を打ちにくい。

一方で近年AIエージェントが急速に発達

依存の管理そのものをAIエージェントに支援させれないか？（本研究）

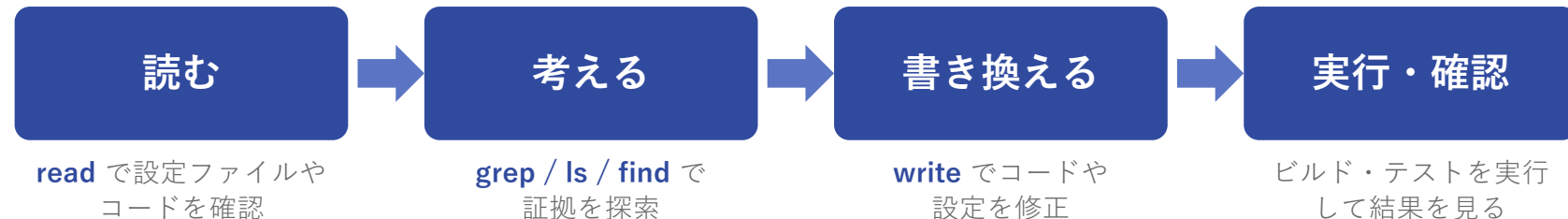
用語：AIエージェントとは？

対話型AI（例：ChatGPT）

- 質問に自然言語で回答する（1回のやり取りが基本）
- ツール操作は補助的で、人が都度指示することが多い

AIエージェント

- 自律的にファイルを開いて読み、コードを書き、テストを実行する
- 結果を見て修正を繰り返す = 人間の開発者に近い働き方



ひうまくいくまで自律的に繰り返す(ループ)

本研究で記録した“エージェントの行動”

対話ターン数

= 操作結果を受け取り、次の行動を返した回数

ツール実行回数

= read / grep / ls / find / write などのツール呼び出し回数

研究の問い（RQ）： AIエージェントは、依存関係の管理をどこまで支援できるか？

位置づけ：実現可能性を確認する初期的ケーススタディ

RQ1：発見・認識

リポジトリ（コード・設定・テスト・文書一式）を丸ごと渡し、外部ライブラリ依存を列挙させる

→ どのように認識するのか

探し方は指定しない

RQ2：内部化

発見した依存を1件ずつ除去し、機能を標準ライブラリ・自前実装・**vendoring** で内部化

→ ビルド／テストが通る状態まで戻せるか

ケーススタディ：Flask を対象に2つの条件で比較

対象:Flask

Python製の有名Webフレームワーク

236ファイル/貢献者732人/コミット5,537件

テスト・文書・サンプル・CI設定を完備

条件A:宣言駆動(マニフェストあり)

- pyproject.toml 等、依存を宣言するマニフェストファイルを直接参照できる
- 手がかかりが豊富な高可視状態

条件B:痕跡駆動(マニフェストなし)

- マニフェストファイルをすべて削除した状態で実験
- import文・テスト・文書に残る痕跡だけから推定させる

マニフェストとは？

プロジェクトが依存するライブラリやバージョンを、機械が読み取れる形式で記述した設定ファイル。

依存関係の「一覧表」の役割を果たす。

例：Pythonのマニフェスト（pyproject.toml 抜粋）

```
[project]
name = "sample-project"
dependencies = [
    "flask>=3.0.0",
    "requests>=2.31.0",
    "jinja2>=3.1.0",
]
```

この一覧が見えるのが条件A。
すべて削除して見えないのが条件B。

なぜ2条件で比較？ マニフェスト（依存の宣言）の有無が、AIの依存発見・内部化にどう影響するかを切り分けるため

実行環境： ChatGPT 5.4 + piハーネス（操作ログを全記録） / 期間 約2週間・費用 約300ドル

手順(RQ1)： リポジトリを渡し、依存を探索・報告させる

リポジトリを丸ごと渡し、AIが自力で依存を見つけて整理する



何を「証拠」とみなすかはAI任せ

手順(RQ2)：1件ずつ削除して検証する

削除目的の別セッションで依存を再探索し、報告された削除対象を1件ずつ内部化する



プロンプトA = 除去 + 内部化の指示 / プロンプトB = 失敗箇所の修復指示

検証に失敗したら → 修復(プロンプトB)を実行

1周 = 依存リストを一巡。

全2周 (1周目：除去 + 内部化 → 2周目：残った失敗を通るまで修復)

RQ1の結果：見える情報で報告数も行動も変わる

宣言駆動(マニフェストあり)

114 件の依存を報告

痕跡駆動(マニフェストなし)

28 件の依存を報告

AIエージェントの行動	宣言駆動	痕跡駆動
対話ターン数	10	24
ツール実行回数	45	97
うち grep(検索)	11	45

「正しい依存リスト」は一つに定まらない。可視情報の有無で発見範囲・行動が変わる
(宣言**114**件／痕跡**28**件、探索行動 約**2**倍)

RQ2の結果：対象範囲が広がると初回通過率が下がる可能性

同じリポジトリでも削除対象が変わる

宣言駆動 43件 / 痕跡駆動 21件。（RQ1とは独立したセッション）

1周目で一発成功した割合は

宣言駆動

46.5%

痕跡駆動

81.0%

(1周目でビルド&テストまで通った割合)

一発成功率 = 初回に修復なしで通った割合

宣言駆動では推移的・文書／ビルド用まで対象に入り、範囲が広がる

→ **宣言駆動では削除対象の範囲が広がり、結果として1件あたりの初回成功率が低下した可能性がある。**

依存の役割別の正解集合が未整備のため、この広がりが有益な網羅なのか、過剰な選択なのかは判断できない。

RQ2の結果：ビルド&テストは通過、ただし機能等価は未確認

指標(2周目の修復まで含む)	宣言駆動	痕跡駆動
総実行時間	3.97 時間	1.40 時間
対話ターン数	1,068	403
入力トークン(AIが読んだ量)	約253万	約69万
2周目の修復が必要だった件数	23 件 / 43 件	4 件 / 21 件

両条件とも、報告された削除対象について「ビルド&テストが通る状態」には到達（＝機能等価は未検証）

ただし一発では終わらず、修復の繰り返しが必要

可視情報が多い宣言駆動条件では対象数と対象の種類が広がり、追加修復が多く、総コストも大きい

考察：4つの知見

見える情報が探索範囲を決める

何を証拠とみなすかで報告される依存が変わる

発見は目的に依存する

独立セッションでも、目的が変わると報告対象が変わる

可視情報が対象の範囲と種類を変える

宣言駆動（可視情報が多い条件）ほど対象範囲が広がり、内部化コストが増える。範囲の妥当性は今後の課題（層別の正解集合が必要）

再実装は修復プロセス

一発生成では終わらない。失敗からの回復効率も含めて評価すべき

本研究の限界と今後の課題

本研究の限界

- Flask 1リポジトリのみ(実現可能性の確認が目的)
- 正解データが未整備
- テスト通過 $\neq$ 意味的に等価(未検証の挙動差があり得る)
- 単一のモデル・ハーネス構成
- 人間の続行操作を挟む半自動設計

今後の課題

- 複数リポジトリ・多言語エコシステムへ拡大し、実現可能性から一般性の検証へ
- 正解データを整備し、発見の網羅性・削除の過剰性を定量評価。テスト通過を超えた意味的等価性まで検証
- 複数モデル/ハーネスでの比較と、人手介入(続行操作)を減らした自動化・コスト最適化
- 内部化すべき依存の選定基準の確立
- 内部化による長期的な便益と新たな負担の評価