# 設計崩壊をさせないAI活用実装プロセスの提案

構造解析ツールをガードレールとして活用した、実装方法の改善

深瀬

テクマトリックス株式会社
ソフトウェアエンジニアリング事業部
www.techmatrix.co.jp

# 自己紹介

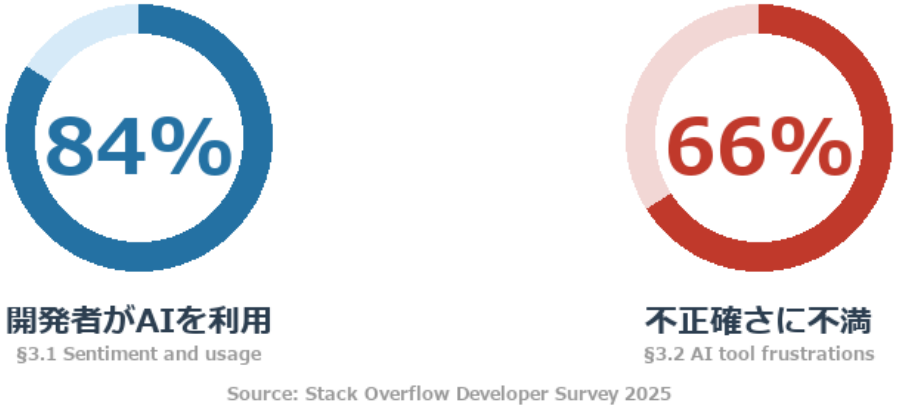
- 深瀬
- テクマトリックス株式会社
- ソフトウェア品質・構造解析ツールの技術支援
- 担当製品: Lattix（DSMベースのアーキテクチャ解析）、Understand（ソースコード解析）

# 本日の内容

- 背景・課題
- 仮説: 構造解析ツールをガードレールとして活用できないか
- 検証1: 小規模サンプルでの是正
- 検証2: 小規模サンプルでの予防
- 検証3: 大規模プロジェクトでの是正
- 検証から得られた知見・提案
- 今後の展望・まとめ

# 背景

- ソフトウェア開発におけるAI活用が急速に普及 ※1
- 既存システムへの機能追加・仕様変更時にAIを利用する機会が増加 ※2
- AIに設計ルールを指示しても、指示通り正しく実装されない場合がある ※2
- AIによるコーディングでソースコードが肥大化し、全体構造の見通しが悪化 ※3




Stack Overflow Developer Survey 2025

※1 Stack Overflow Developer Survey 2025 §3.1, §3.2

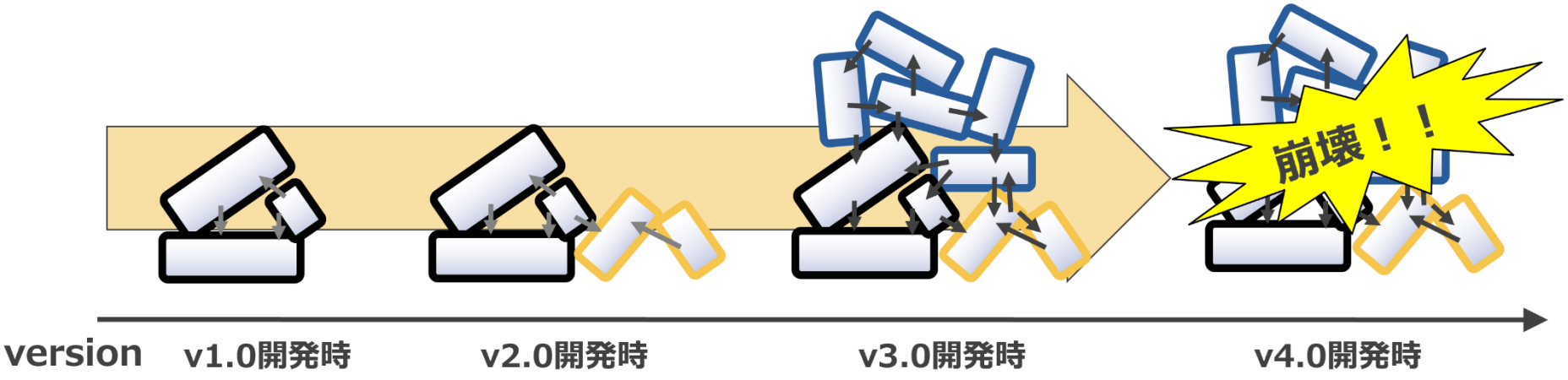
※2 Liu, M. et al. (2026) arXiv:2603.01814

※3 Zhu, Y. et al. (2026) arXiv:2605.02741

※ 番号は巻末の参考文献と対応

# 課題: AI実装による構造劣化

- 構造劣化は従来の人手開発でも発生していた
- 生成AIはコード生成速度が速い分、構造劣化の蓄積も速い
- 人手による修正には時間的・工数的な限界がある
- → 設計を乱さずにAI活用を進める仕組みが必要




構造劣化のイメージ

# なぜAIは設計ルールを守らないのか

| 原因 | 説明 |
| --- | --- |
| 最短経路志向 | AIは仕様を最短で実現しようとする ※4 |
| 暗黙ルールの無視 | レイヤー構造、依存方向の制約は<br>明示しなければ無視される ※5 |
| 俯瞰の欠如 | コンテキスト内の情報だけで判断し、<br>全体構造を見ない ※2 |
| 制約増加で性能低下 | 構造的制約が増えるほどAIの遵守率は急落する<br>（Constraint Decay） ※4 |

※4 Dente, L. et al. (2026): タスクを最短で達成する実装を優先し、制約が増えるほど遵守率が急落すると報告. arXiv:2605.06445

※5 Jiang, Z.; Nam, J. (2026): OSSリポジトリのCursor Rules調査で、プロジェクト固有ルールの明示が必要と指摘. Proc. MSR 2026. arXiv:2512.18925

※2 Liu, M. et al. (2026): リポジトリ全体の構造を把握できない「architectural blindness」を指摘. arXiv:2603.01814

# 仮説と本発表のスコープ

## 仮説

**構造解析ツールの設計ルール情報をAIにフィードバックすれば、AIによる設計ルール違反を予防・是正できるのではないか**

## 本発表のスコープ

| プロセス | 本発表で検証すること | 検証しないこと |
|---|---|---|
| 是正 | 既にある設計ルール違反を、AIに修正させられるか | 人のレビューを介さない全自動の是正<br>個別の設計ルールの是非 |
| 予防 | 追加実装時に、AIに違反を作らせずに済むか | 設計ルールそのものの妥当性評価 |

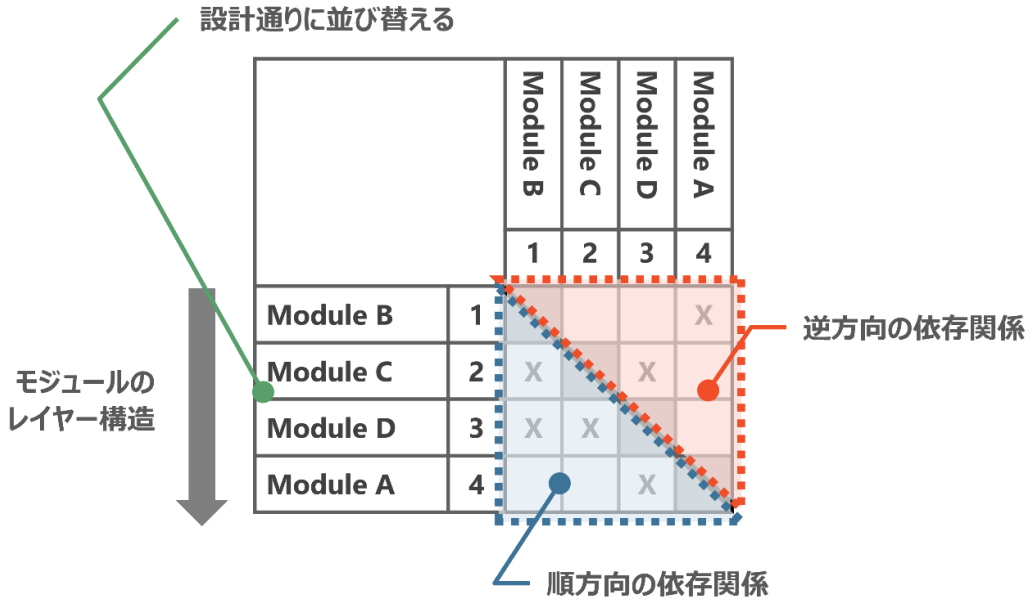
# 実験条件: 使用したツール

| ツール | 役割 |
| --- | --- |
| GitHub Actions | 解析～違反検出の自動実行 |
| GitHub Copilot | AIによるコード生成・修正（Agent Mode） |
| Lattix | Understandの解析結果を入力し、DSMでアーキテクチャを可視化・ルール違反を検出 |
| LattixWeb | バージョン間で違反数の推移を比較 |
| Redmine | AIが作成した修正設計案をチケットとして登録・レビュー |
| Understand | ソースコードの依存関係を解析 |

※ GitHub Actions・LattixWeb・Redmineは大規模の検証3で使用

# 補足: 本発表で使用するDSMの見方

- DSM（依存構造マトリクス）: モジュール間の依存関係をマトリクスで可視化
- 左下 = 順方向の依存（正常）
- 右上 = 逆方向の依存（設計上問題） → ルール違反の対象




DSMの見方

# 補足: 本発表で使う用語

- **本発表では、これらのレポートをAIへのフィードバックに用いる**

| 用語 | 説明 | AIに渡すか |
|---|---|---|
| 設計ルール | レイヤー間の依存の可否の定義<br>例: server → modules 禁止 | 定義のみ<br>（渡さない） |
| 設計ルールレポート | 設定した設計ルールの一覧 | **予防**で渡す |
| 設計ルール違反 | 設計ルールに反する依存関係 | 定義のみ<br>（渡さない） |
| 設計ルール違反<br>レポート | 検出した設計ルール違反の一覧<br>違反となる依存関係の依存元、<br>依存先、依存の種類 | **是正**で渡す |

# ガードレールという考え方

- 構造解析ツール（Lattix）をガードレール（安全装置）として使えるのではないか
- 考えられる活用方法は次の2つ
- いずれもLattixの設計ルール情報がAIへのフィードバックの起点になる
- この2つが実際に機能するかを検証する

| アプローチ | 役割 | 概要 | タイミング |
|---|---|---|---|
| ① 是正 | **ガードレール**<br>逸脱を検知して戻す | 事後に違反を検出しAIに修正させる | コード生成後 |
| ② 予防 | **ハーネス**<br>そもそも逸脱させない | 事前にルールを渡して違反を防ぐ | コード生成前 |

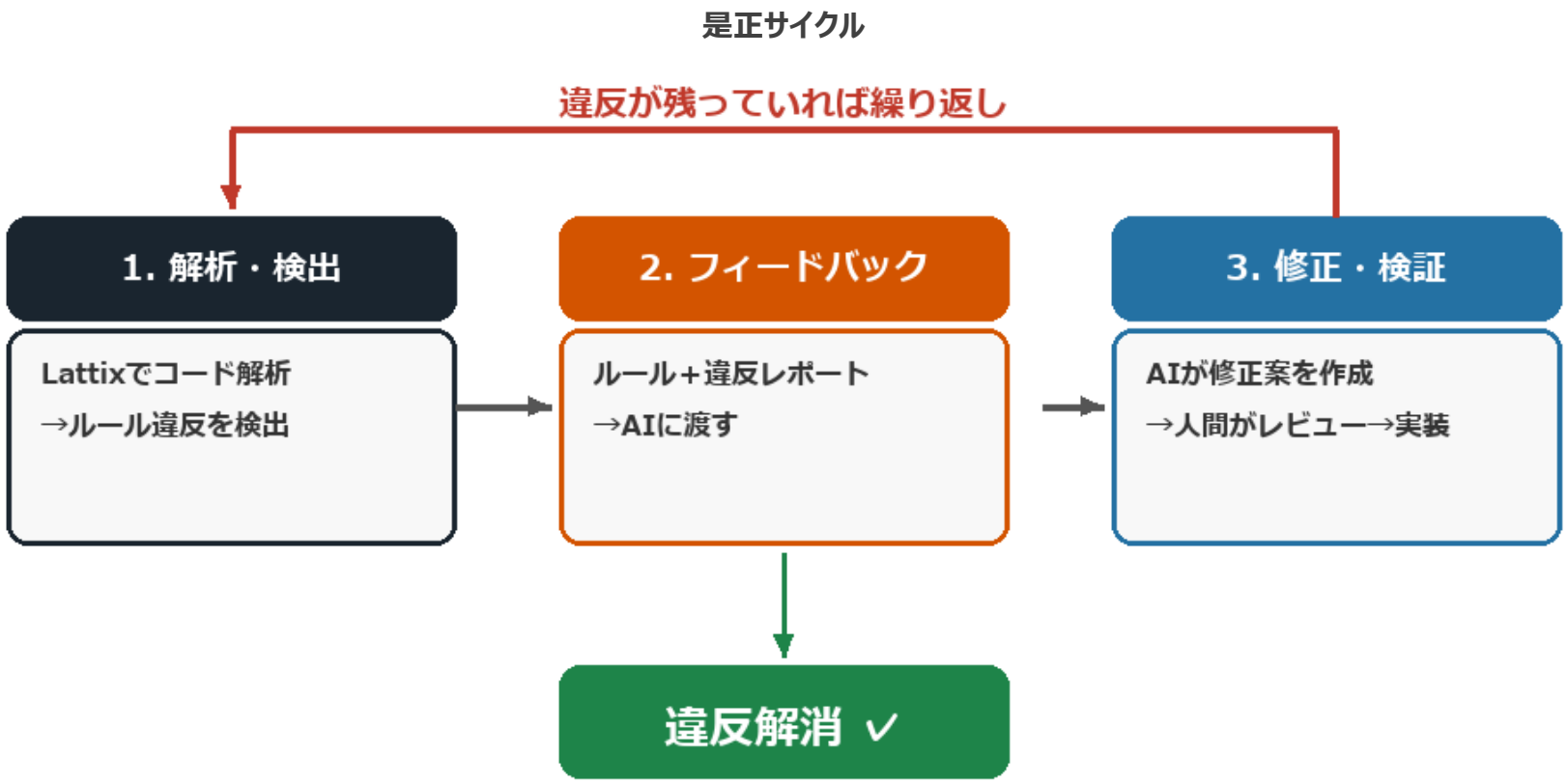
# 検証の計画

- **2つのアプローチが実際に機能するかを、規模を変えた3ケース（いずれもC言語）で検証する**
- **まず小規模で是正と予防を確かめ、そのあと大規模で確かめる**

| 検証 | アプローチ | 検証内容 | 対象（規模） |
|---|---|---|---|
| 検証1 | 是正 | **既存の違反**をAIが直せるか | 3層サンプル<br>65行 |
| 検証2 | 予防 | **追加実装時の違反**を<br>防げるか | 組み込み<br>ファームウェア<br>約200行 |
| 検証3 | 是正 | **大規模**でも通用するか | Apache httpd<br>約40万行 |

# アプローチ①: 是正サイクル

- 既にルール違反が発生している場合に適用するプロセス




ツールがAIの設計品質をチェックし、AIがコードを修正する —— このサイクルを繰り返す

# 検証1: 既存の違反をAIが直せるか

| | 内容 |
|---|---|
| アプローチ | ① 是正（コード生成後に違反を検出しAIに修正させる） |
| 問い | 設計ルール違反レポートと設計ルールレポートの2つを渡すだけで、AIは修正できるか |
| 対象 | app / view / model の3層サンプル（C言語、65行） |

# 検証1: 検証の条件と結果

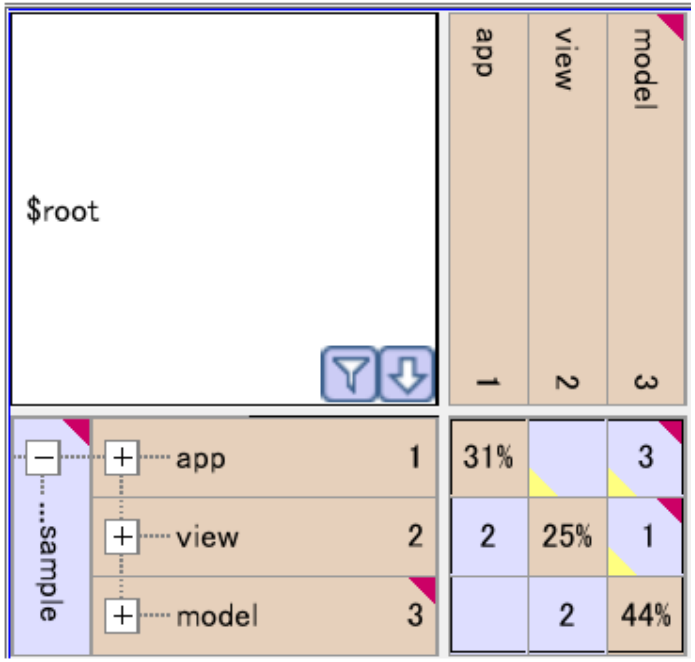
- **AIは2つのレポートを読み取り、変数定義の移動・インクルードの修正を実施**

| 項目 | 内容 |
|---|---|
| 対象 | app / view / model の3層サンプル（65行） |
| 設計ルール | **3本**（view→app / model→app / model→view を禁止） |
| 初期違反 | model→app, model→view の逆依存 4件 |
| AIへの入力 | 設計ルール違反レポート + 設計ルールレポート |
| 使用ツール | Lattix 2025.1.7 + Understand 7.0 |
| AIモデル | Claude Haiku 4.5（GitHub Copilot） |
| 結果 | 4件すべて解消（違反0件） |

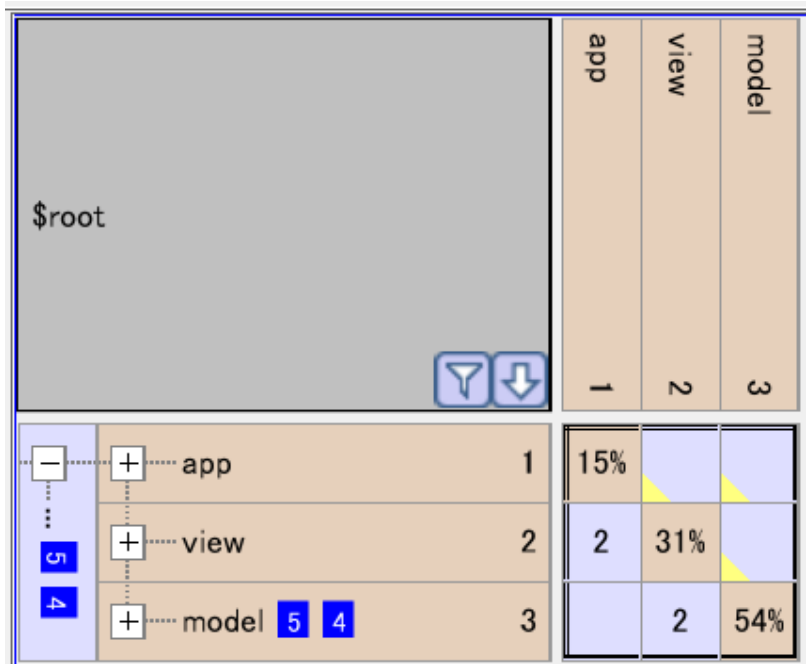
**今回は2つのレポートを渡すだけで、AIは4件すべての違反を修正できた**

※ 検証1だけAIモデルとUnderstandのバージョンが異なるのは実施時期の違いによる。当時社内で利用できたAIモデルがClaude Haiku 4.5のみだった

# 検証1: DSM修正前後の比較




修正前（4件の逆依存）




修正後（違反0件）

逆依存を4件とも解消し、DSM上の違反が0件になった

# アプローチ②: 予防プロセス

- 新たに機能を追加する際に適用するプロセス



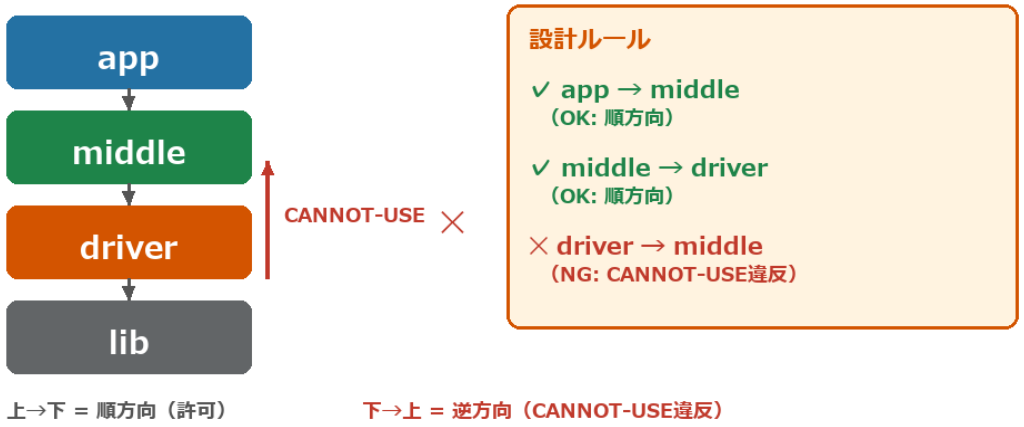

コード生成の「前」にルールを渡すことで、違反そのものを発生させない

# 検証2: 追加実装時の違反を防げるか

| | 内容 |
|---|---|
| アプローチ | ② 予防（コード生成前にルールを渡して違反を防ぐ） |
| 問い | ルールを事前に渡すだけで、違反の発生を防げるか |
| 対象 | 組み込みファームウェア（C言語、4層、約200行） |

# 検証2: 検証の条件

| 項目 | 内容 |
|---|---|
| 対象 | 組み込みファームウェア（C言語、4層、約200行） |
| レイヤー | app / middle / driver / lib |
| 追加仕様 | UART割り込み駆動化 |
| 設計ルール | **12本**（CAN-USE 6本 + CANNOT-USE 6本）<br>うち今回注目するのは driver → middle: CANNOT-USE |
| 使用ツール | Lattix 2025.1.7 + Understand 7.2 |
| AIモデル | Claude Opus 4.6（GitHub Copilot Agent Mode） |




4層レイヤー構造と設計ルール

# 検証2: ルールの有無による違い

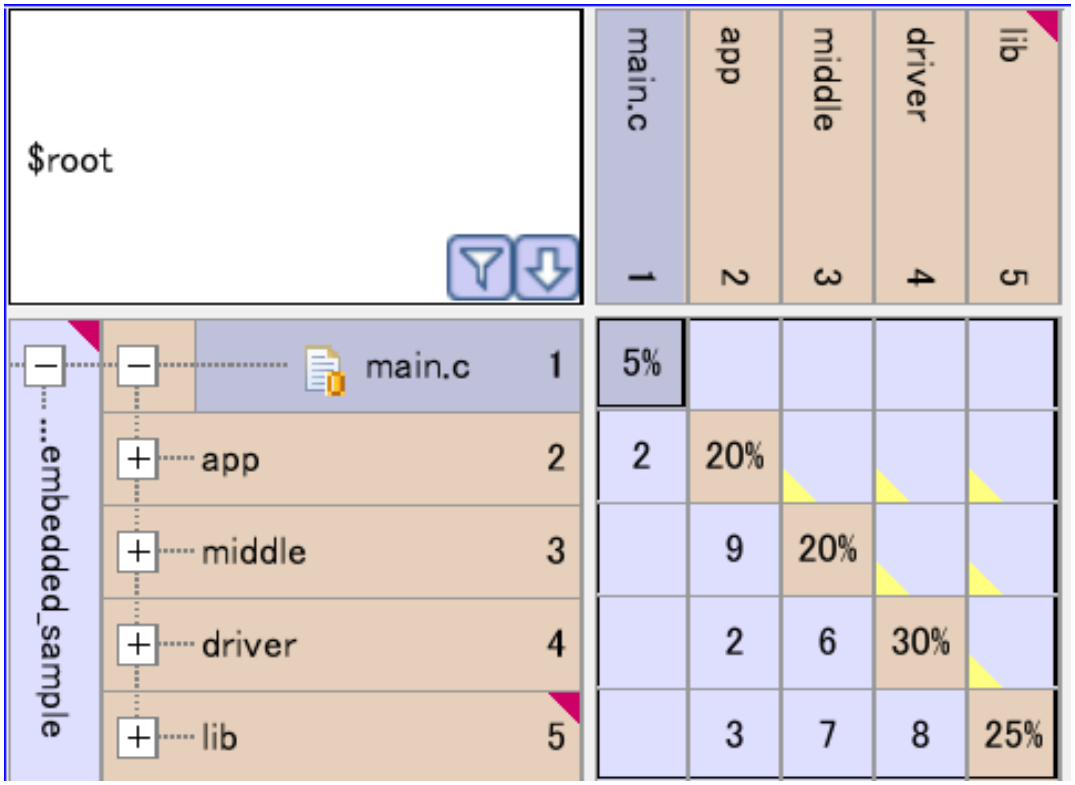
| | パターンA（ルールなし） | パターンB（ルールあり） |
|---|---|---|
| AIへの入力 | 追加仕様書のみ | 追加仕様書 ＋<br>設計ルールレポート |
| AIの設計判断 | driver→middle直接呼出 | driver層内にバッファ配置 |
| 依存構造 | driver→middleの逆依存 | driver層内で完結 |
| ルール違反 | 2件（driver→middle） | 0件 |

**ルールを事前に渡すだけでAIの設計判断が変わり、違反の発生を防げた**

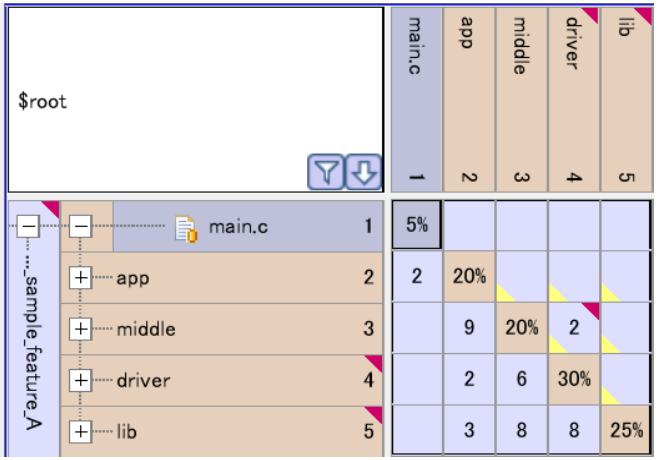
※ パターンA: driver層の割り込みハンドラがmiddle層のバッファ管理関数を直接呼び出す設計

※ パターンB: driver層内にローカルバッファを配置し、middle層への依存を回避する設計

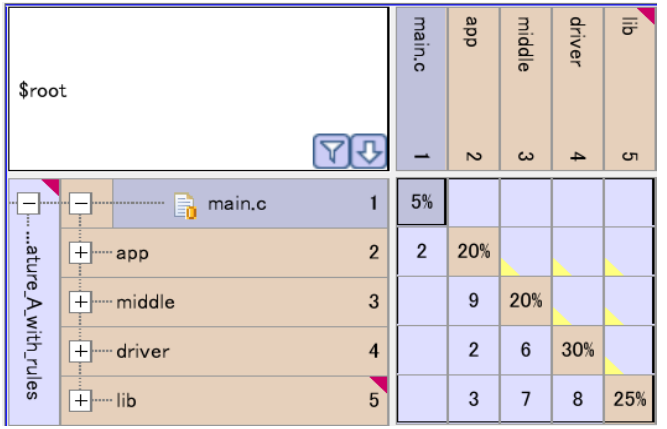
# 検証2: DSMで見る設計判断の違い




実装前




パターンA（違反2件）




パターンB（違反0件）

ルールの有無だけで、生成されるコードの依存構造が変わった

# 中間まとめ: 小規模では是正も予防も機能した

- **しかしこれは、冒頭で挙げた「AIは指示通りに実装しない」「制約が増えるほど遵守率が急落する」とは異なる結果**
- **対象が小規模（65行 / 約200行）で、違反が関わる範囲も限られていたためと考えられる**

| 検証 | やったこと | 結果 |
|---|---|---|
| **検証1**<br>是正 | 設計ルール違反レポートと<br>設計ルールレポートを渡す | 逆依存4件を<br>**すべて解消（違反0件）** |
| **検証2**<br>予防 | 実装前に<br>設計ルールを渡す | **違反0件**<br>（渡さない場合は2件発生） |

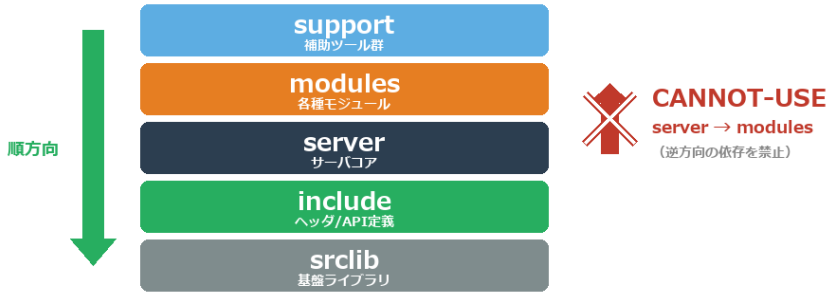
**大規模プロジェクトでも同じことが言えるのかを確かめる**
**→ Apache httpd（約40万行）で検証**

# 検証3: 大規模でも通用するか

| | 内容 |
|---|---|
| アプローチ | ① 是正（コード生成後に違反を検出しAIに修正させる） |
| 問い | 大規模プロジェクトでも是正サイクルは機能するか |
| 対象 | Apache httpd 2.0.55（C言語、約40万行） |

# 検証3: 大規模コードでの検証環境

| 項目 | 内容 |
|---|---|
| 対象コード | Apache httpd 2.0.55（C言語、約40万行） |
| レイヤー | support / modules / server / include / srclib |
| 設計ルール | **1本**（server → modules: CANNOT-USE） |
| 使用ツール | Lattix 2025.1.7 + Understand 7.2 |
| AIモデル | Claude Opus 4.6（GitHub Copilot Agent Mode） |

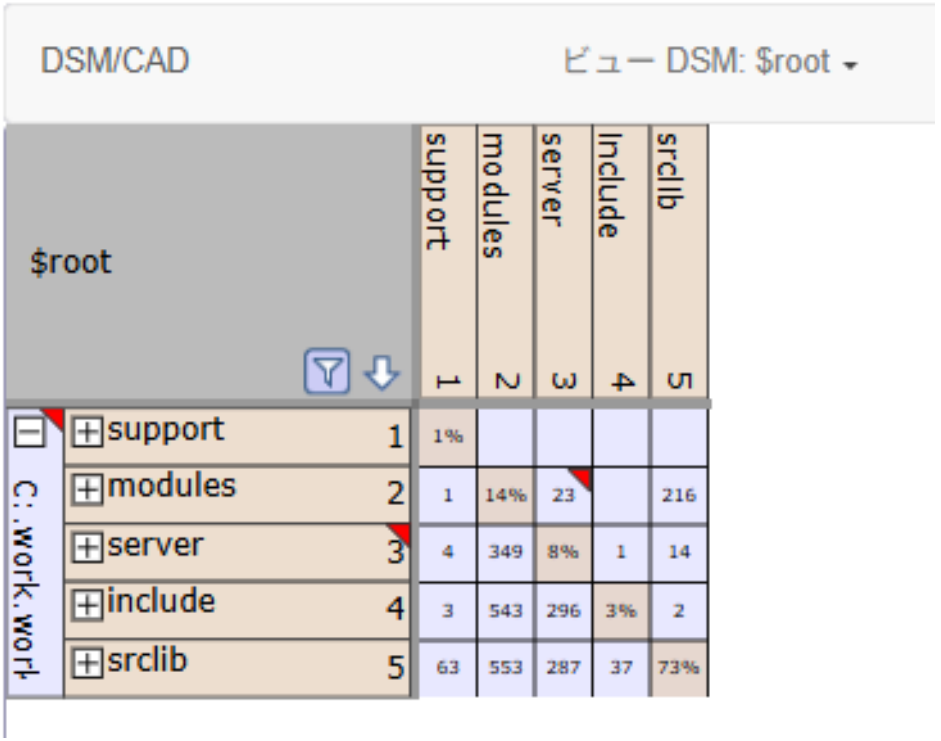



Apache httpdのレイヤー構造と設計ルール

※ 逆依存は他のレイヤー間にもあったが、多数あったため今回の修正対象を絞る目的で設計ルールを1本に限定した

# 修正前のDSM — 31件の違反がある状態

- server → modules の赤セル = 設計ルール違反（逆依存）

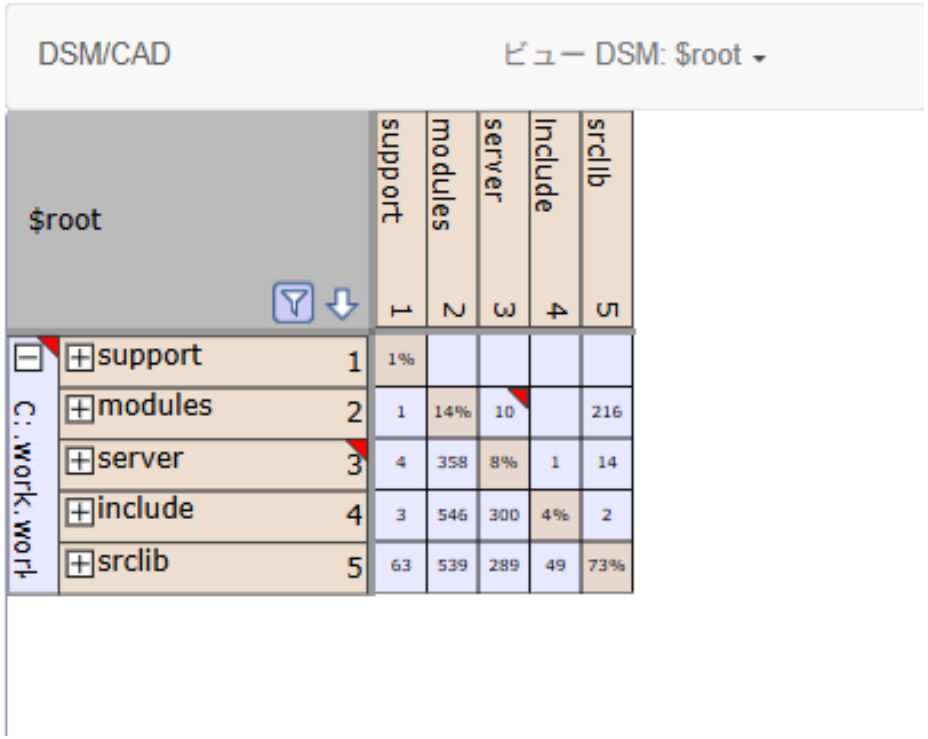



修正前のDSM（31件（※）の違反）

※ カウントの仕方がDSMと設計ルール違反レポートで異なる。
DSMは依存のあるファイルペア、設計ルール違反レポートはそのファイルペアを依存関係の種類別にカウントしているため、31件=23ペアとなる

# 修正後のDSM — 18件に減少した状態

- 是正後、赤セルが減少（31件 → 18件）




修正後のDSM（18件（※）に減少）

※ カウントの仕方がDSMと設計ルール違反レポートで異なる。
DSMは依存のあるファイルペア、設計ルール違反レポートはそのファイルペアを依存関係の種類別にカウントしているため、18件＝10ペアとなる

# 検証3で実際に行った是正の手順

- Lattixの設計ルール違反レポートをAIに渡し、AIが修正設計案を作成
- 設計案はRedmineチケットとして登録し、人間がレビュー・承認してから実装
- 実装後はLattix再解析で違反解消を確認、AIが単体テストも作成・実行



etc #25988 未完了

Violation Evidence Group 004: server/protocol.c -> modules/http/http_request.c (Invoke: ap_die)

さんが [2026-08-12 09:26] 15日 前に追加.

**ステータス:** New
**優先度:** Normal
**担当者:** -
**カテゴリ:** -
**作業者:**

**説明**

- アーキテクチャルール違反として指摘された依存関係
  依存元: server/protocol.c / 依存先: modules/http/http_request.c / 件数: 1 / 依存関係の種類: Invoke / ルール: server CANNOT-USE modules
  （ルール出典: ldcreport -violations 出力 violations_25982.csv）

違反詳細（証跡）:

- server/protocol.c:988 → ap_die(access_status, r)

**是正設計案**

方針: ap_die およびその依存関数群（ap_internal_redirect, ap_internal_redirect_handler, ap_allow_methods + static helpers 3件）を modules/http/http_request.c から server/request.c へ移動する。

server層配置の根拠:

- 呼び出し元がserver/protocol.c, server/request.c
- 依存先が全てserver層（ap_invoke_handler, ap_process_request_internal, ap_finalize_request_protocol等）
- リクエストライフサイクルのエラーハンドリング・内部ルーティング機能
- AP_DECLAREで公開されたコアAPI

1. modules/http/http_request.c から7関数を削除
2. server/request.c に7関数を追加（static helpers含む）
3. ヘッダ宣言(include/http_request.h)は変更不要
4. und analyze + ldcreport で違反解消を確認

Redmineチケットの例

# 修正設計案8件がすべて成功

- **8件の修正設計案をすべて実装し、Lattix再解析で計13件の違反解消を確認**

| No. | 解消数 | AIの修正パターン |
|---|---|---|
| No.1 | 1件 | 関数移動 → server/request.c, core.c |
| No.2 | 1件 | 共通関数抽出 → include/ |
| No.3 | 4件 | 関数9つを移動 → server/protocol.c, request.c |
| No.4 | 1件 | 関数移動 → server/protocol.c |
| No.5 | 1件 | 関数移動 + 追加修正（Lattixが新違反を検出） |
| No.6 | 3件 | ヘッダ移動 → mod_core.h を include/ へ |
| No.7 | 1件 | ヘッダ移動 → mod_proxy.h を include/ へ |
| No.8 | 1件 | グローバル変数の定義移動 |

**AIが8件すべてを修正し、Lattixで違反の解消を確認。修正成功率100%（8/8）**

# 違反31件→18件に減少（削減率41.9%）

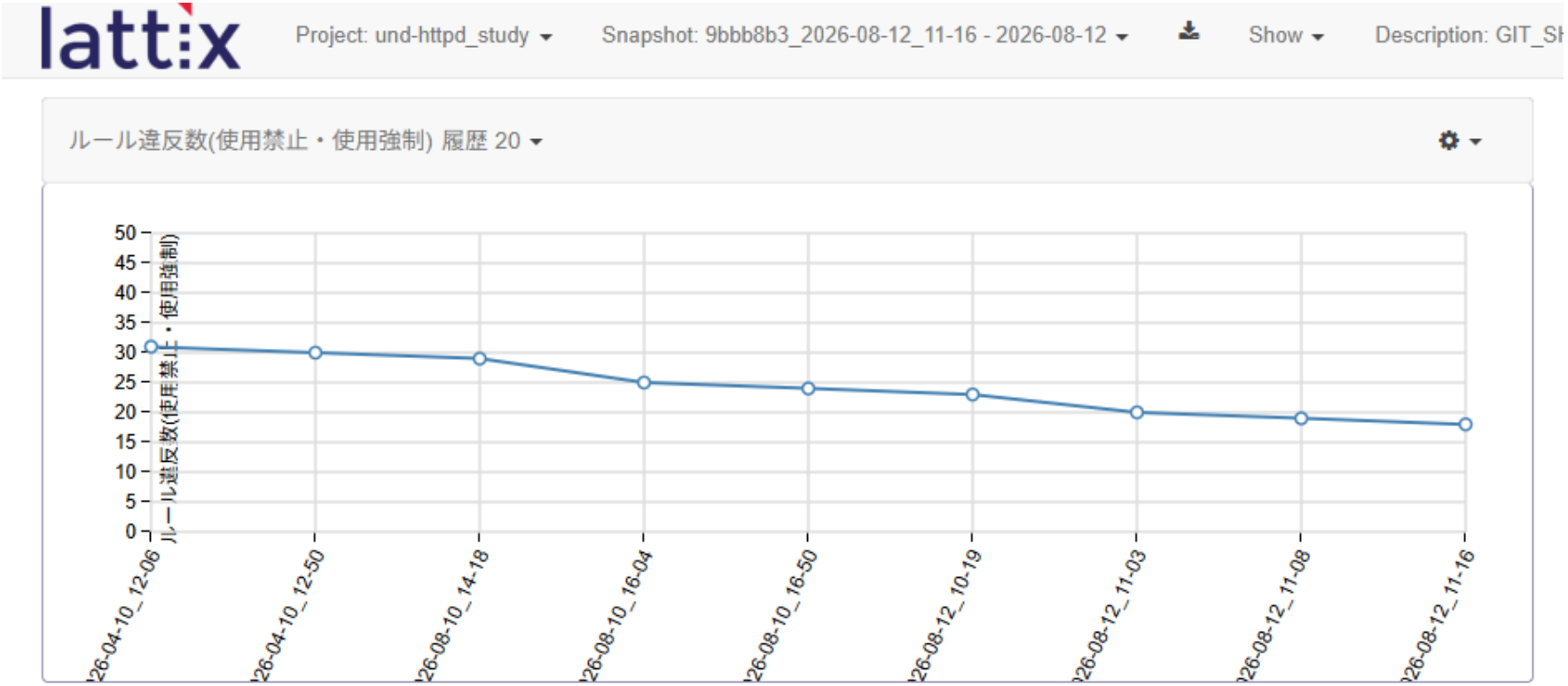
- 残り18件は、AIが「修正は適切ではない」と判断したもの。その判断の妥当性を人が個別に検証するのは本検証のスコープ外のため、人の判断でここで打ち切った
- 削減率 = 解消した違反数 ÷ 初期違反数（31件）

| 時点 | 違反数 | 解消数 | 累計削減率 |
|---|---|---|---|
| 初期状態 | 31件 | — | — |
| No.1実装後 | 30件 | 1件 | 3.2% |
| No.2実装後 | 29件 | 1件 | 6.5% |
| No.3実装後 | 25件 | 4件 | 19.4% |
| No.4実装後 | 24件 | 1件 | 22.6% |
| No.5実装後 | 23件 | 1件 | 25.8% |
| No.6実装後 | 20件 | 3件 | 35.5% |
| No.7実装後 | 19件 | 1件 | 38.7% |
| No.8実装後 | 18件 | 1件 | 41.9% |

**大規模コードでも是正サイクルが機能し、AIに任せることで違反を削減することができた**

# 違反数の推移をLattixWebで追跡

- LattixWebのバージョン比較機能で違反数の推移をグラフ表示




LattixWebによる違反数の推移

# 検証3: AIの修正が正しいとは限らない

- ここまで8件の修正設計案がすべて成功したことを示した
- しかし、是正の過程ではAIが誤った修正を行い、
  新たな違反を生んだケースもあった
- 修正設計案No.5で実際に起きた事例を紹介する

# 検証3: AIの誤りをガードレールが検出した事例

- **Apache httpd の修正設計案No.5で何が起きたか**

| 段階 | 内容 |
| --- | --- |
| ① AIが修正 | 8つの関数をmodules層からserver層に移動 |
| ② Lattix再解析 | 元の違反は解消、しかし新たな違反が1件発生 |
| ③ 原因 | 移動した関数が呼んでいる先がまだmodules層に残っていた |
| ④ フィードバック | 新違反の内容をAIに伝える |
| ⑤ 追加修正 | 呼び出し先の関数もserver層に移動 → 違反解消 |

※ 具体的な関数名・ファイル名レベルの詳細はAppendix参照

# 修正設計案1つあたり平均16分で解消

- 修正設計案1つあたりの平均: 約9ターン、約16分（No.1を除く7件）
- No.5は初回修正で新違反が発生し、追加修正を行ったため時間が伸びた

| No. | 解消数 | AIターン数 | 純粋作業時間 |
|---|---|---|---|
| No.1 | 1件 | 2ターン | 約6分 |
| No.2 | 1件 | 10ターン | 約20分 |
| No.3 | 4件 | 18ターン | 約25分 |
| No.4 | 1件 | 8ターン | 約10分 |
| No.5 | 1件 | 16ターン | 約40分 |
| No.6 | 3件 | 5ターン | 約10分 |
| No.7 | 1件 | 3ターン | 約5分 |
| No.8 | 1件 | 4ターン | 約5分 |

※ AIターン数 = 指示1回+応答1回を1ターンとし、設計案作成・実装・テスト作成を合計した回数

※ 純粋作業時間 = 設計案作成～Lattix再解析で違反解消の確認まで（待ち時間・環境デバッグを除く）。集計の都合上、単体テスト作成は含まない

※ No.1は設計案作成をGitHub Actionsで自動実行しており、他の7件と工程が異なるため平均から除外

# 修正の正しさを2段階で確認

- **①でルール違反が解消されたことを確かめ、②で振舞いが変わっていないことを確かめる**
- **単体テストもAIが自動生成（38件、内訳はAppendix参照）**

| 段階 | 確認した内容 | 確認方法 | 結果 |
|---|---|---|---|
| ① 構造的正しさ | ルール違反が解消し、新違反もないこと | Lattixで再解析 | 13件解消 |
| ② 機能的正しさ | 関数のロジックが変わっていないこと | 単体テスト（関数を実行して確認） | 38件全Pass |

# 検証3の事例から得られた教訓

- **構造解析ツールのフィードバックがなければ、この新違反は見逃されていた**
- **→ AIを信用するのではなく、ツールで検算する仕組みがガードレールとして機能する**

| 観点 | 教訓 |
| --- | --- |
| AIの限界 | AIは指示された違反は直せるが、修正が波及して生じる別の依存までは見通せない |
| ツールの役割 | 修正後の再解析を必ず行うことで、人間の目では見逃す可能性がある新違反を検出できる |
| プロセス | 「修正→再解析→フィードバック」を**1回で終わらせずループさせる**ことが必須 |

# 検証結果のまとめ

- **仮説: 構造解析ツールの設計ルール情報をAIにフィードバックすれば、違反を予防・是正できるのではないか**

| 検証の問い | 対象規模 | 結果 |
|---|---|---|
| **検証1（是正）**<br>既にある違反を直せるか | 65行 | 4件すべて解消 |
| **検証2（予防）**<br>違反を作らせずに済むか | 約200行 | 違反0件 |
| **検証3（是正）**<br>大規模でも通用するか | 約40万行 | 13件解消（削減率41.9%）<br>設計案8件成功 |

**仮説は部分的に支持された。是正は小規模・大規模、予防は小規模までの範囲**

※ 大規模での予防は未検証（今後の宿題）

※ 残存課題: 大規模では41.9%を削減したものの、31件中18件（58.1%）が未解消のまま残っている

# 検証を踏まえた提案

- **構造解析ツールをガードレールとして活用するプロセス**
- **予防では制約が増えるほどすり抜けが出ると考えられる（冒頭で示したConstraint Decay）**

| プロセス | やること | AIに渡す情報 |
|---|---|---|
| **① 是正**<br>コード生成後 | 違反を検出してAIに修正させ、<br>**修正→再解析→フィードバック**<br>をループさせる | 設計ルール違反レポート<br>＋ 設計ルールレポート |
| **② 予防**<br>コード生成前 | 設計ルールを渡してから<br>実装させる | 設計ルールレポート |

**予防で防ぎきれないものを、是正でガードレールすることが重要**

# 検証から得られた知見

- AIは「動くコード」を書くのは得意だが「設計意図」は伝えなければ汲み取らない
- 約40万行のコードでも是正サイクルが機能する（8件全成功）
- AIが誤った修正をしてもツールが検出しフィードバックできる
- 小規模ではプロンプトに一文添えて設計ルールレポートを渡すだけでAIの設計判断が変わった（予防）
- 構造解析ツールはAI出力が設計ルール（依存関係）を守れているかを定量的に検証する手段として有効
- 一方、AIが「この違反は設計意図に沿ったもので直すべきではない」と判断した事例もあった
- → AIの設計判断の妥当性をどう検証するかは今後の課題

※ AIに渡したのは、是正では設計ルール違反レポート+設計ルールレポート、予防では追加仕様書+設計ルールレポート

# 今後の展望

| テーマ | 内容 |
| --- | --- |
| **大規模での予防の検証** | 今回予防を確かめたのは約200行・設計ルール12本の場合のみ。規模と制約が増えたときにどの程度防げるかを確かめる |
| CI/CD統合 | GitHub Actionsでの解析・違反検出の自動化<br>（デモ環境レベルでは構築済み） |
| ハーネスエンジニアリング | 今回は人間がチャットで逐一指示・承認していた。（検証3#1を除く）<br>AIが自律的に是正サイクルを回せる実行環境を整備する |
| 他のプロジェクトへの適用 | 他言語（Java, C++等）の他のプロジェクトへの適用<br>設計品質レベル別コードへの適用 |
| 振舞いの担保 | 機能テストを実施し、是正によるバグ混入の有無を検証 |

# まとめ

| | 小規模サンプル | 大規模プロジェクト |
|---|---|---|
| 規模 | 65行 / 約200行 | 約40万行（Apache httpd） |
| **是正** | 逆依存4件を<br>すべて解消 | 13件解消（削減率41.9%）<br>設計案8件すべて成功 |
| **予防** | 違反0件<br>（ルールなしでは2件） | **未検証（今後の宿題）** |

**考察: 小規模では予防が効いたが、大規模コードでは制約が増えるほどAIは指示どおりに動かなくなる可能性が考えられる（冒頭で示したConstraint Decay）**
**したがって予防だけに頼らず、すり抜けを是正でガードレールすることが要になる**

**構造解析ツールをAIへのフィードバックに使うことは、設計崩壊の緩和や改善に有効**

※ 今回変えたのはコード規模であり、制約（設計ルール）の数を変えた検証は行っていない

※ 大規模の是正では31件中18件（58.1%）が未解消のまま残っている

# Appendix: 検証結果の定量データ

| 指標 | 定義 | 結果 |
| --- | --- | --- |
| 修正設計案 | AIが作成した修正設計案の数 | 8件 |
| 修正成功率 | Lattixで違反解消を確認できた割合 | 100%（8/8） |
| 解消した違反数 | Lattixで消えたルール違反の件数 | 13件（31→18） |
| 削減率 | 解消違反数 ÷ 初期違反数 | 41.9%（13/31） |
| 単体テスト | AI生成テストのケース数とPass率 | 38件、全Pass |
| 修正設計案1つあたりの作業時間 | 設計案作成～違反解消確認の作業時間 | 平均約16分 |

# Appendix: 単体テスト38件の内訳

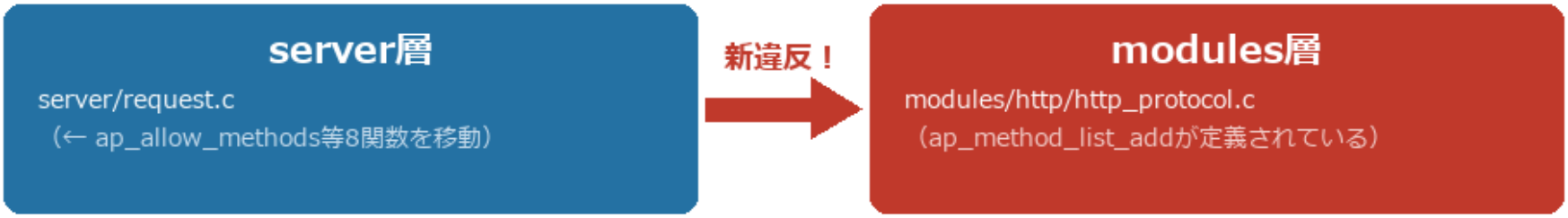
- 対象関数をビルド不要な形で切り出し、実行して振舞いを確認

| 検証内容 | 確認したこと | 件数 |
| --- | --- | --- |
| 動的なロジック検証 | 関数を実際に実行し、入力に対する出力が移動前と同じか | 38件、全Pass |

※ AIはこのほかに、関数の定義位置や宣言の有無をパターンマッチで確かめる静的な構造検証を104件生成しているが、振舞いを確かめるものではないため件数から除外している

# Appendix: 新違反の検出（詳細）

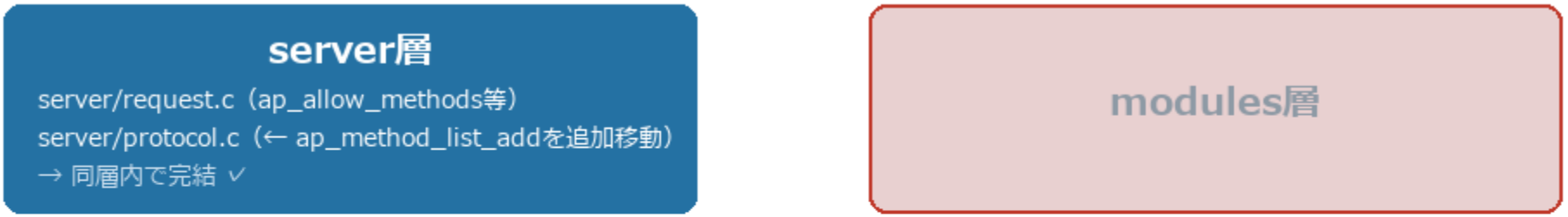
- AIはap_allow_methods等8関数をserver/request.cに移動した
- しかしap_allow_methodsは内部でap_method_list_addを呼んでいた
- ap_method_list_addの定義はmodules/http/http_protocol.cに残っていた
- → Lattixの再解析で依存関係
  server/request.c → modules/http/http_protocol.c
  の新違反を検出




ap_allow_methodsが内部でap_method_list_addを呼んでいた
→ 移動した関数の呼び出し先がまだmodules層に残っていた
Lattixの再解析でこの新たな違反を検出 → AIにフィードバック

# Appendix: 追加修正での解消（詳細）

- Lattixの設計ルール違反レポートをAIにフィードバック
- AIがap_method_list_addをserver/protocol.cに移動する追加修正を実施
- → Lattix再解析で違反解消を確認




✓ ap_method_list_addをserver/protocol.cへ移動 → 違反解消

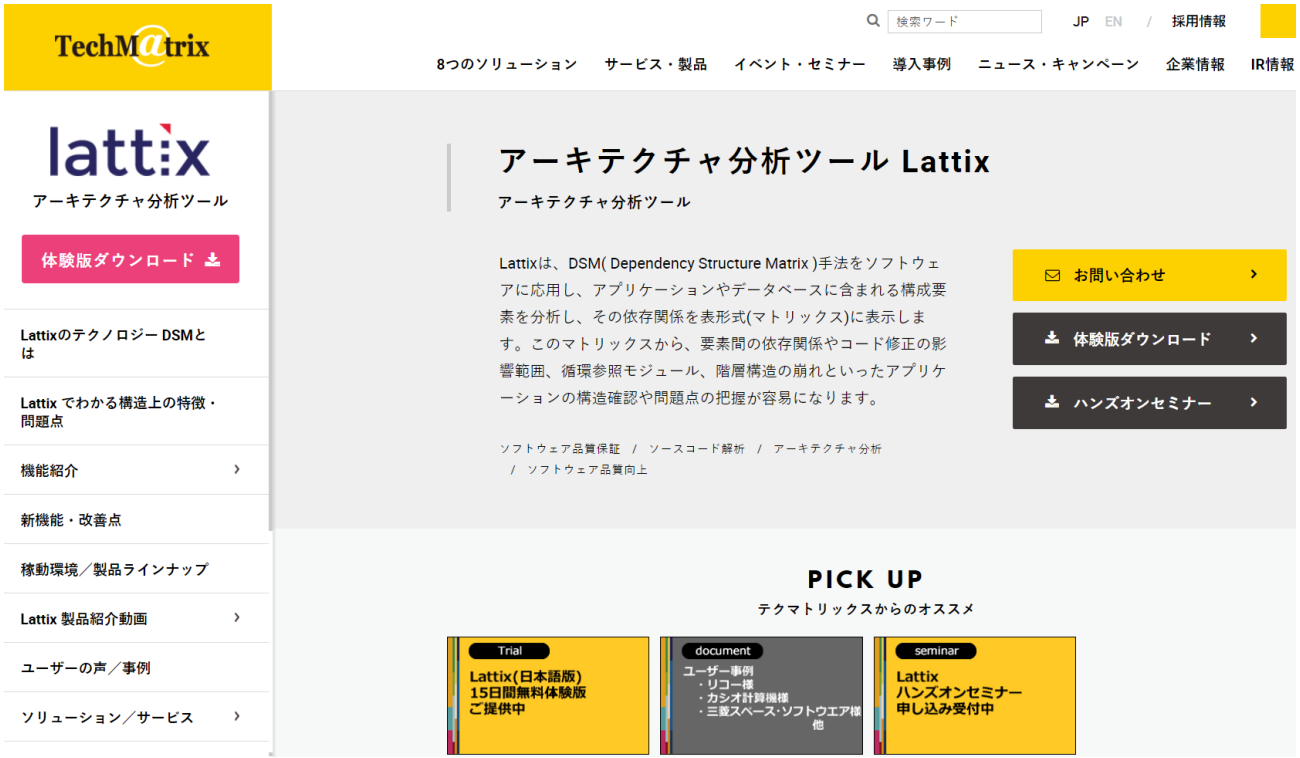
構造解析ツールのフィードバックがなければ、この違反は見逃されていた

追加修正で解消

# 参考文献


| # | 名称 | URL |
|---|---|---|
| 1 | Stack Overflow Developer Survey 2025 | https://survey.stackoverflow.co/2025/ |
| 2 | Architecture-Aware Multi-Design Generation (Liu et al., 2026) | https://arxiv.org/abs/2603.01814 |
| 3 | AI-Generated Smells: An Analysis of Code and Architecture (Zhu et al., 2026) | https://arxiv.org/abs/2605.02741 |
| 4 | Constraint Decay: The Fragility of LLM Agents in Backend Code Generation (Dente et al., 2026) | https://arxiv.org/abs/2605.06445 |
| 5 | Beyond the Prompt: An Empirical Study of Cursor Rules (Jiang & Nam, MSR 2026) | https://arxiv.org/abs/2512.18925 |
| 6 | Lattix アーキテクチャルール機能 | https://www.techmatrix.co.jp/product/lattix/function/architecturerule.html |
| 7 | Lattixで指摘したルール違反を生成AIに修正させてみた | https://understand.techmatrix.jp/lattix_rule-violations-report_in-aidd/ |
| 8 | LattixのルールをAIに渡して実装させてみた | https://understand.techmatrix.jp/lattix_rule-constrained-coding_in-aidd/ |

# お問い合わせ




Lattixの詳細は[こちら](www.techmatrix.co.jp/product/lattix/)

www.techmatrix.co.jp/product/lattix/


TechMatrix

テクマトリックス株式会社
ソフトウェアエンジニアリング事業部

03-4405-7853
lattix-info@techmatrix.co.jp
www.techmatrix.co.jp/product/lattix/