投稿日 2025/03/26

インフラコードに対するAIコードレビュー：GitHub CopilotとPromptisの活用

はじめに

IaC（Infrastructure as Code）開発における品質保証は、安定したインフラ環境を維持するために非常に重要です。本記事では、GitHub CopilotとPromptisを活用したコードレビュープロセスの改善について、実際のフィードバックと改善を解説します。

IaC（Infrastructure as Code）について

IaCとは「Infrastructure as Code」の略で、インフラストラクチャをコードとして管理・構成する手法です。従来の手動によるサーバー設定やネットワーク構成作業を、プログラミング言語やテキストファイルで定義することで自動化します。

代表的なIaCツール

1. **Terraform**: クラウドリソース管理に特化した宣言型ツール
2. **Ansible**: シンプルなYAML構文で構成管理を行うツール
3. **AWS CloudFormation**: AWSリソース専用のテンプレート形式
4. **Azure Resource Manager (ARM) テンプレート:** Azure専用のJSONベースのテンプレート

AIコードレビュー：利用者フィードバック分析

以下はGitHub CopilotとPromptisを活用したIaCコードのAIレビューについて利用者からのフィードバックをまとめたものです。利用・フィードバック分析は TIS IT基盤技術企画部のプロダクトチームと共同で実施しました。

1. フィードバックの概要

- IT基盤技術企画部 中西氏から提供されたフィードバック資料をもとに議論が行われた
  - 利用参加者はインフラエンジニアの毛利氏、福島氏、宮田氏の3名

    - 利用ツールと言語は、Terraform、Ansible、Python
    - 主な実施範囲：非機能観点のコードレビュー、コード標準のレビュー
- 全体的にポジティブな評価が得られており、利用者の反応は良好
    - AIによるコードレビューは、「セルフレビューの強化」として有効
    - 得られたフィードバックより、使用プロンプト、Promptisをそれぞれ改善し、再評価を実施した

2. フィードバックからの改善

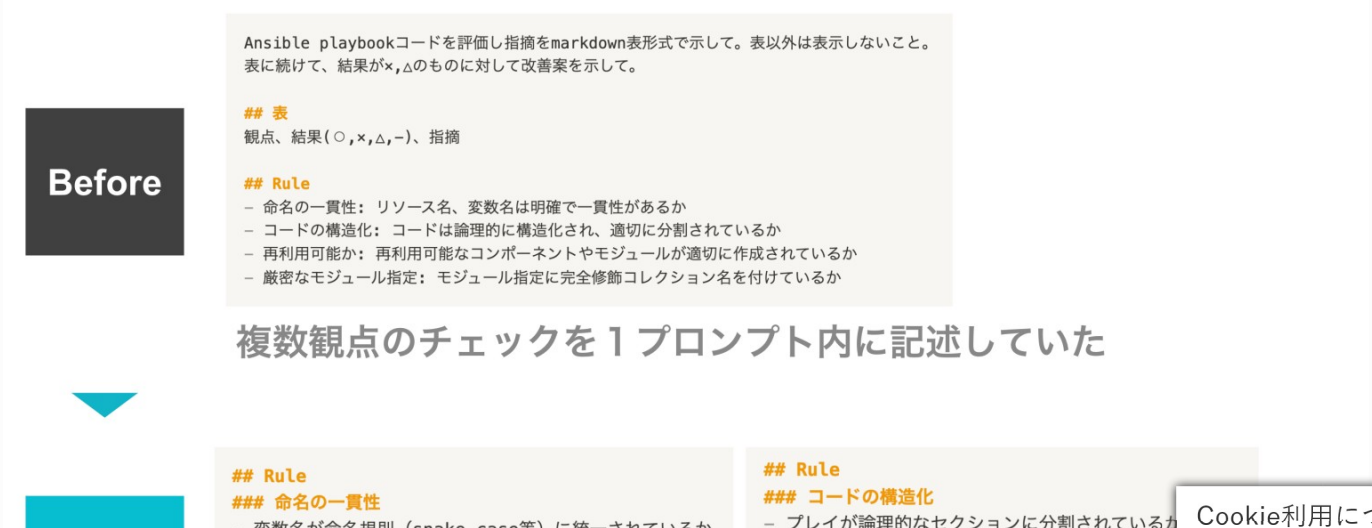
① 使用プロンプトの改善
② Promptisの改善
③ 効果的な活用のための利用者への認識合わせ

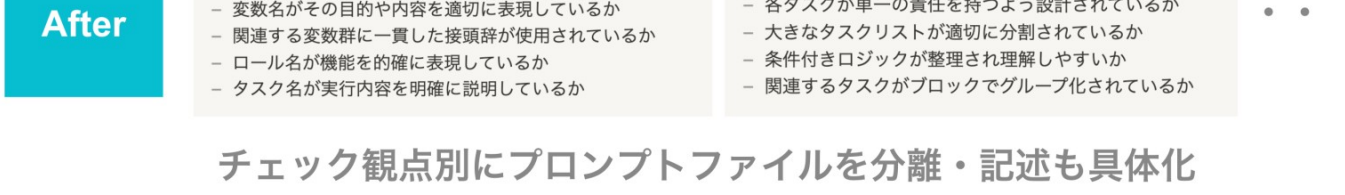
①使用プロンプトの改善

一部のプロンプトでは、実行するたびに結果が微妙に異なる一貫性の問題が報告されました。命名規則のチェックに対し大文字/小文字の誤った指摘や、出力フォーマットの不安定性（表形式と箇条書きの混在）が一部みられました。対策として以下のプロンプトの改善を行いました。

プロンプト設計の改善

1. **シンプルさを重視する**
    - 文字数を少なくすることが最も効果的
        - 箇条書きは5-6個程度に抑えるのが効果的
        - 入力の観点が多すぎると、個別の出力結果が省略されてしまう傾向
        - 敬語調を使わず体言止めを用いて文字数を削減
2. **チェック観点を分離する**
    - 例.命名規則のチェックとその他チェックは別プロンプトとして独立させる
        - AIにも「認知負荷」があり、観点を限定させた方が精度向上する
        - 大文字/小文字、接頭語のルールチェックは特に独立させ、細かく指示をするように変更
    - 人間にとっても同種のチェックをまとめた方が確認しやすく、プロンプトファイルを保守しやすい





②Promptisの改善

ユーザーフィードバックを受け、Promptisに対し 2 つの改善を行いました。

1.（機能追加）ディレクトリ指定での一括実行

改善におけるメリット

- 複数ファイルに対し連続してレビューを行うことが 1 コマンドで実施できるようになった
- 実行ログ内でどのファイルに対する出力結果が見やすくなり、該当箇所の特定が容易に(例.パス違いの同名ファイルの判別)

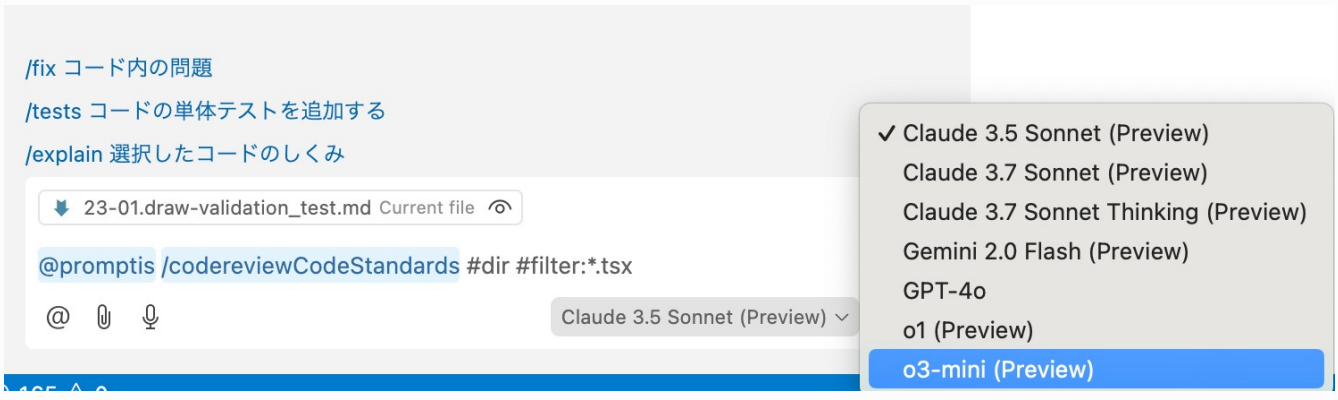
チャット変数	説明	例
#dir	プロンプト中に#dirを含めることで、プロンプトを適用するディレクトリを指定できる。指定したディレクトリ配下の全ファイルに対してプロンプトが実行される。	@promptis /codereviewCodeStandards #dir

2.マルチモデル選択と使い分け

Promptisは使用するLLMモデルを固定としていましたが、実行時にユーザーがモデルを選択できるようにしました。

改善におけるメリット

- Claude 3.7やGPT-o3miniなどの新しいモデルの使用による精度が大幅に向上
- 専門家レベルのユーザビリティチェックや深い分析が可能となった
- GitHub Copilotの定額料金モデルは、APIベースのトークン課金と比較して実験的なチェックがしやすい
    - モデルごとの出力の差異や傾向、最適なプロンプト改善を繰り返し実施しやすい



③ 効果的な活用のための利用者への認識合わせ

利用者側のコツとして、特にAIによる自動チェック結果の解釈方法についてチーム内の認識を揃えました。

AIによるチェック結果の解釈方針

- 「○」の項目は鵜呑みにせず、「△/×」の項目を重点的に確認
- AIによるチェック結果を確認材料として活用し、最終判断は人間が行う
- 品質のゲートキーパー（砦）ではなく、ゲートウェイ/フィルターとして扱う

Review Details

- Prompt: ../python/prompt/code_standard/01-03.Ansible-code-standards-check.md
- Target: lamp_haproxy/aws/roles/base-apache/tasks/main.yml

表

観点	結果 (○,×,△,-)	指摘
機能が適切にロールとして分離されているか	○	
ロールやタスクが適切にパラメータ化されているか	△	パッケージ名やサービス名がハードコーディングされている
コンポーネントが様々な環境で使用できるよう設計されているか	△	SELinuxの設定が特定の環境に依存している
再利用のための使用方法が文書化されているか	×	使用方法が文書化されていない
コンポーネント間の依存関係が明示的に定義されているか	○	
コンポーネントが個別にテスト可能な設計になっている	×	個別にテスト可能な設計になっていない

改善案

ロールやタスクが適切にパラメータ化されているか

```
---
# This role installs httpd

- name: Install packages
  yum:
    name: "{{ item }}"
    state: present
  with_items: "{{ packages }}"
```

AIによるチェック結果の抜粋（Ansibleコードに対するもの）

今後の展開と将来展望

CIとの連携活用

GitHub Copilotを含むローカルツールとCIパイプラインとの組み合わせは効率的なレビュープロセスの実現に有効です。

- **ローカルツール**：定額で何度も試せる、開発中のカジュアルなチェック・早期フィードバック
- **CI連携**：自動化されたチェックによるレビュー効率化
  - マージリクエスト前の最終検証
  - ライブラリ更新時の影響確認
  - 定期的なコード品質チェック・技術負債リスクの早期検出

ツール	主な焦点	タイミング	強み	チェック項目
Linter	コード品質と規約	開発中・コミット前	潜在的バグの検出	• 構文エラー • 未使用変数 • 命名規則 • セキュリティパターン
Formatter	コードスタイルの統一	保存時・コミット前	自動整形による一貫性	• インデント • スペース • 改行 • 括弧の配置
GitHub Copilot	コード検証と改善提案	リアルタイム	パターン認識による検証	• 一貫性 • ベストプラクティス • エッジケース • リファクタリング候補
CI連携 (AIレビュー)	自動化されたレビュー	PR・マージ前	客観的な品質評価	• 一貫性分析 • 品質メトリクス • 影響範囲 • セキュリティ脆弱性

開発段階、実行環境、実行タイミングでAIの活用方法の整理

AIフィードバックの特徴は、多少の精度のばらつきがあっても迅速に返答が得られる点です。製造の初期段階では速度を優先し、ある程度のばらつきを許容します。一方、後の段階では安定性を重視し、必要に応じて複数回のチェックや人間の確認を追加することが効果的です。

開発段階	実行環境	実行タイミング	AIの役割	ねらい
製造中：初期	ローカル	随時	・コードサジェスト	・カジュアルな試行錯誤
製造中：初期	ローカル	随時	・リファクタリング提案 ・セルフレビュー	・何度も繰り返し実行 ・即時フィードバックによる学習効果
製造中：中期	ローカル	コミット前	・具体的な問題解決 ・初期バグ検出	・早期問題発見 ・レビューア負担の軽減
製造中：後期	CI環境	マージ前	・コード品質確認 ・コード標準準拠の検証	・チェック状況をレビューアへ共有 ・品質チェックゲートウェイとしての役割 ・より確実性が求められるチェック

デプロイ前	CI/CD	マージ後	・セキュリティチェック ・整合性確認	・チーム全体への共有 ・レビューアの負担減 ・ チェックよりは警告
保守フェーズ	CI/CD	・定期実行 ・イベント駆動	・潜在的問題の検出 ・技術的負債の特定	・ リスク予防・早期検出 ・ 課題のトリアージ（振り分け）と優先順位付け

アジャイル原則との親和性

AIによる早期フィードバックはアジャイル開発の原則と非常に相性が良いです。「速さ」を重視するアジャイルの価値観にAIの特性が合致し、短いイテレーションでの継続的なフィードバックと改善の文化を強化します。「完璧を目指すのではなく、継続的に改善する」というアジャイルの思想とAIの特性は一致しており、スプリントごとのレビュープロセスにAIを統合することで効果を最大化できます。

開発者体験の向上

従来の人間によるレビューと比較して、AIは圧倒的な速度でフィードバックを提供します。これにより問題が複雑化する前に対処でき、「作りながら学ぶ」という開発スタイルが促進されます。コードの問題を早期に発見することで、後のリファクタリングコストも大幅に削減できます。

即時フィードバックは開発者の学習を促進し、満足度を向上させます。「書いてすぐにレビューを受けられる」という体験は非常に価値があり、開発者はAIとの対話を通じて自己成長の機会を得られます。レビュー待ちによるストレスや遅延が減少することで生産性が向上し、開発者はより創造的な課題に集中できるようになります。

まとめ

GitHub CopilotとPromptisを活用したIaCコードレビューは、効率的な品質保証プロセスの実現に大きく貢献します。特にインフラ構築のような複雑で影響範囲の広い領域では、AIによる自動チェックと人間の専門的判断を組み合わせることで、より安全で高品質なコード管理が可能になります。

TISでは今後もプロンプト設計の改善や複数ファイル対応などの機能拡張を進めることで、さらに効果的なプロセス改善を進めていきます。

/* Recommend */
「Generative AI（生成AI）」 のおすすめ記事はこちら

この記事に関連する記事もお読みください。

Generative AI（生成AI） NEW

OctoNihon Forumイベント発表資料「GitHub Copilotを活用した大規模開発の 今 と 未来」公開

2025/06/06 4

Generative AI（生成AI）

GitHub Copilotを活用した大規模開発 ~オフショア開発での実践と知見~

2025/02/26 16

Generative AI（生成AI）

GitHub Copilotを用いたAIによるコードレビューの活用

2025/01/28 26

最近投稿された記事も用意しました。

Generative AI（生成AI） NEW

「金融業界特化型AIエージェントワークショップ」開催レポート

2025/06/13 8

Generative AI（生成AI） NEW

OctoNihon Forumイベント発表資料「GitHub Copilotを活用した大規模開発の 今 と 未来」公開

2025/06/06 4

ITアーキテクチャ NEW

生成AIの新潮流：AIエージェント勉強会を開催しました

2025/05/02 12

「Generative AI（生成AI）」 で最も読まれている記事を以下にまとめています。

Generative AI（生成AI）

GitHub Copilotを用いたAIによるコードレビューの活用

2025/01/28 26

Generative AI（生成AI） NEW

OctoNihon Forumイベント発表資料「GitHub Copilotを活用した大規模開発の 今 と 未来」公開

2025/06/06 4

Generative AI（生成AI）

GitHub Copilotを活用した大規模開発 ~オフショア開発での実践と知見~

2025/02/26 16