Vol.2025-SE-219 No.13
2025/3/4

RAGを利用したREST API誤用に対する自動修正の有効性調査

井上 翔瑛[1]　山岸 克紀[1]　吉田 則裕[1]　槇原 絵里奈[1]

概要：IoT サービスの多くは REST API を提供しており，クライアント開発者は REST API 仕様に基づきコーディングする必要がある．プログラムに誤用が含まれる場合，多くはエラーレスポンスを基にデバッグするが，原因特定には不十分な場合が多く，サーバとの送受信が多発する．そこで，大量のテキストデータを学習した大規模言語モデル (LLM) による自動修正が考えられるが，ドメイン特有のクエリを扱う際に誤った出力をする可能性がある． これを解決するために，外部情報をプロンプトに含めることで LLM による生成精度を高める手法である RAG (Retrieval-Augmented Generation) が登場した．しかし，既存研究では REST API 誤用の自動修正に有効であるか示されていない．本研究では，REST API 誤用を含むクライアントのプログラムを対象として，関連する仕様書のテキスト片をプロンプトに含めて LLM に与えることで自動修正を行い，その有効性を調査する． 調査実験では，REST API の誤用事例を収集し，単純な LLM の利用と標準的な RAG 手法，およびデータベースを拡張した RAG 手法に適用することで修正率を比較した．その結果，単純な LLM の利用による修正率は 60.5%であったが，標準的な RAG 手法によって 73.5%に高めることができた．さらに，データベースを拡張した RAG 手法では修正率を 80.0%に高めることができた．

An Investigation of RAG-Based Automated Repair for REST API Misuses

Shoei Inoue[1]　Katsuki Yamagishi[1]　Norihiro Yoshida[1]　Erina Makihara[1]

1. はじめに

近年，IoT デバイスがますます普及しており [1], [2]，API を活用した開発手法に注目が集まっている [3]．IoT デバイスのクライアント開発では計算リソースの制約，多様なネットワークプロトコルの利用，エンティティの統合といった要素を考慮する必要がある [1]．これにより，従来のシステムよりも複雑性が増すため，開発時に多くのバグが発生しやすい状況にあり，発生しているバグが十分に修正されていないことが問題となっている [4]．

IoT サービスの多くは, REST (Representational State Transfer) アーキテクチャスタイル [5] に基づいたウェブ API である REST API を提供している．そのため，開発者はクライアント側からリソースに対して容易にアクセスすることができる．クライアントは，REST API を提供するプロバイダが管理するリソースに対して，HTTP リクエストを送信することでアクセスを行う．このとき，アクションを示す HTTP メソッド（例：GET や POST）とリソースのパスを指定することで，必要なサービスを利用できる．特に，GET や POST といったメソッドでは，リクエストにパラメータやコンテンツなどの追加情報を含めることがある．また，リクエストの結果は，ステータスコードとしてクライアントに返され，成功可否が通知される．

REST API のプロバイダが提供するリソースにアクセスするためには，公開されている REST API 仕様書に基づいてコーディングする必要がある．そのため，クライアント開発者は仕様書に記述されている REST API 仕様に従わなければならない．一般的には，仕様通りに正しく記述でき

[1] 立命館大学
Ritsumeikan University

ているか確認するために HTTP リクエストを送信し，レスポンスを受信してその結果を確認する．しかし，レスポンスに含まれるエラーコードやエラーメッセージは情報が少なく，エラーの発生箇所を特定する内容は書かれていない場合が多い．そのため REST API のクライアント開発において，開発者は HTTP リクエストの送信とレスポンスの受信を繰り返しながらデバッグを行うことが多い．

また，REST API の仕様変更が行われた場合，開発者はクライアント側のソースコードを新しい仕様書に基づいて修正しなければならない．しかし，開発者が修正を行わないままになっているものが数多く存在しており，脆弱性などの欠陥を引き起こす原因となる [6], [7]．そこで，開発者にはソースコードを早急に最新の仕様へ適応させる修正が求められる．

これまで，Java の API を対象とした誤用検出 [8], [9] や GPT-3[10] などの事前訓練済みの大規模言語モデル (LLM) を活用したソースコードの自動修正 [11], [12], [13], [14] に関する研究が行われてきた．これらの研究では，Java の API 仕様を満たさないメソッド呼び出しを検出および修正することを目的としている．しかし，Java の API は REST API とはクライアントにおける呼び出し方法が大きく異なる．具体的には，Java の API では API を呼び出すメソッド呼び出しが仕様を満たす必要がある．一方，REST API ではエンドポイントやリクエストヘッダ，リクエストボディが仕様を満たす必要がある．

我々の研究グループでは，REST API を対象とした LLM による自動バグ修正手法 [4] を提案している．この手法では，REST API 仕様とクライアントのソースコードから誤用箇所を検出して，LLM へ入力することで自動修正を行うアプローチを取っている．しかし，先行研究ではプロンプトに誤用箇所を含めているものの，修正に必要な情報は含まれていない．そのため，LLM が学習していない REST API 仕様に対して正確な修正ができないという欠点がある．

そこで，RAG (Retrieval Augmented Generation)[15] に着目した．RAG は，特定のドメインに特化した情報や，修正に必要な情報を含めることができる特徴を持ち，先述の課題を解決することが可能だと考えたためである．本研究では，REST API を呼び出すクライアントのプログラムを対象に，標準的な RAG 手法およびデータベースを拡張した RAG 手法を適用し，単純な LLM の利用と比較することで性能を調査した．

調査実験では，IoT サービスの REST API を提供している SwitchBot と Fitbit に対する REST API 誤用事例を収集し，調査手法を適用した．まず RQ1 として，単純な LLM の利用と標準的な RAG 手法の精度を調査したところ，修正率が 13.0%向上した．次に RQ2 として，標準的な RAG 手法とデータベースを拡張した RAG 手法の精度を調査したところ，修正率が 6.5%向上した．

表 1: SwitchBot API の仕様[*1]

Parameter	Type	Location	Required
Authorization	String	header	Yes
sign	**String**	**header**	**Yes**
t	**Long**	**header**	**Yes**
nonce	**Long**	**header**	**Yes**

以降，2 章で本研究に関する用語や背景について述べ，3 章で調査手法や実験に用いるデータセット，および評価方法を述べる．4 章で実験結果を述べ，5 章で関連研究を述べた後に 6 章で本研究のまとめと課題点を述べる．

2. 背景

2.1 REST API

IoT デバイス開発の多くは，REST (Representational State Transfer) アーキテクチャスタイル [5] に基づいたウェブ API である REST API を採用している．そのため，開発者はクライアント側からリソースに対して容易にアクセスできる．REST API を利用するクライアントはリクエストを送信する際，必要なパラメータを含める必要がある．必要なパラメータとは，主に以下の 4 つである．

- パスパラメータ : リソースを指定する URI
- クエリパラメータ : リソースを取得する情報
- リクエストヘッダ : リクエストに関する付加情報
- リクエストボディ : リソースの追加および更新情報

IoT サービスの REST API として SwitchBot API と Fitbit API が挙げられる．

SwitchBot API[16] とは，SwitchBot 社が提供している API であり，SwitchBot を用いてスマートロックや照明など IoT 機器の操作を行うことができる．SwitchBot API は 2022 年に REST API の一部バージョン変更を行い，新製品は旧バージョンの仕様を非推奨にしている．**表 1** の細字はバージョン 1.0 で追加され，太字はバージョン 1.1 で追加された REST API 仕様の一部である．仕様変更によってリクエストヘッダに新しく sign，t，nonce が追加されている．

Fitbit API[17] とは，Fitbit 社が提供している REST API であり，ウェアラブルデバイスを通してフィットネス情報の記録や取得ができる．Fitbit API は正確なバージョン変更の時期を公開していないものの，OAuth1.0 から OAuth2.0 への移行や一部のバージョン 1.2 への変更に伴い，旧バージョンの仕様を非推奨にしている．

SwitchBot は GitHub[16] で，Fitbit は開発者向け公式ページ [17] で REST API 仕様が公開されている．クライアント開発者は，公式ページが提供している REST API 仕様を基に開発を行う．

図 1 は，SwitchBot API の非推奨仕様に基づいた誤用事

[*1] 太文字はバージョン 1.1 で追加された REST API 仕様の一部
[*2] https://github.com/snoozers/lazy-home/commit/bb9a929d723cc51b67cf08a09de2d183406b3569

```
- BASE_END_POINT = 'https://api.switch-bot.com/v1.0'
+ BASE_END_POINT = 'https://api.switch-bot.com/v1.1'

+ def headers() -> dict:
    {中略}
+   return {
+     'Authorization': token,
+     't': str(t),
+     'sign': str(sign, 'utf-8'),
+     'nonce': nonce
+   }

  devices = requests.get(
    url=BASE_END_POINT + '/devices',
-   headers={
-     'Authorization': os.environ['SWITCH_BOT_OPEN_TOKEN']
-   }
+   headers=headers()
  ).json()['body']
```

図 1: SwitchBot API の誤用事例[*2]と修正例

例と，その修正例である．文の先頭にマイナスを記述しているのが誤用を含むプログラムで，文の先頭にプラスが記述しているのが修正されたプログラムである．誤用は 1 行目のエンドポイント，および 15 行目から 17 行目のリクエストヘッダである．リクエストヘッダは Authorization だけでなく，sign，t，nonce を記述する必要がある．修正例では，2 行目にエンドポイントを修正し，4 行目から 11 行目にかけてリクエストヘッダを返却する関数を作成している．ここでは，リクエストヘッダに新しく追加された sign，t，nonce があり，正しく動作するコードに修正されている．このように，エンドポイントだけでなくリクエストヘッダも変更される仕様変更があるため，どちらも正しく修正する必要がある．

2.2 LLM を用いた既存の自動バグ修正手法および問題点

LLM を用いた自動バグ修正手法を 3 種類取りあげる．

1 種類目は，単純な LLM の利用である．Xia らは，既存の自動プログラム修正ツール [18] と事前訓練済みの LLM によるバグ修正率を比較している [11]．Xia らが行った研究は，LLM を自動バグ修正に直接適用しており，9 つの LLM を用いて，3 つの異なるプログラミング言語にまたがる 5 つのデータセットへ適用することで評価を行っている．LLM の評価を行うにあたり，修復設計を用いて LLM の自動バグ修正に対する性能を調べている．以下に修復設計を示す．

- 関数全体の生成 : バグを含む関数全体のパッチを生成する．
- 正しいコードの挿入 : 関数の接頭辞と接尾辞を指定して，パッチを生成する．
- 単一行の修正 : バグを含む行のみパッチを生成する．

これらの修復設計とデータセットを用いて LLM によるバグ修正個数などを評価している．Xia らは，LLM が従来の自動プログラム修正ツールと比較して，特定のプログラミング言語に依存しない汎用性や高精度でプログラムを自動修正できる能力を有していることを示した．一方で，REST API 仕様などの外部情報を必要とするバグへの有効性は評価されていない．

2 種類目は，ファインチューニングを用いた手法である．Li らは，事前訓練済みの LLM に対して，一部のパラメータを効率的にファインチューニングする Parameter-Efficient Fine-Tuning (PEFT) 手法を提案している [19]．Li らが行った研究は PEFT 手法と，既存手法のモデル全体をファインチューニングする Full-Model Fine-Tuning (FMFT) 手法を比較することで有効性と効率性を検証することを目的としており，3 つのベンチマークを用いて評価を行っている．この手法は，PEFT 手法と比較して，少ない計算資源で性能を維持および向上できる点に強みがある．一方で，修正性能は LLM のベースモデルに依存しているため，ファインチューニングする前に最適なベースモデルを選択する必要がある．また，Li らはパラメータとデータセットのサイズによって修正性能は顕著に増加しないことを示した．したがって，修正性能を高めるために何度も微調整をしなければならない問題がある．

3 種類目は，我々の研究グループが提案している，プログラムの誤用箇所を検出してプロンプトを拡張する手法である．この手法は，REST API 仕様とクライアントのプログラムを照合することで誤用箇所を検出し，それをプロンプトに含めることで，LLM によるバグの修正率を高めることを目的としている [4]．REST API 仕様を誤って記述しているプログラムの箇所を特定できる点に強みがあり，LLM による自動バグ修正の精度向上が期待できる．具体的には，プログラムの構文木を REST API 仕様と照合し，仕様を満たしていないプログラムの誤用箇所と，REST API 仕様の一部をプロンプトのテンプレートに埋め込むことで実現している．ただし，クライアントのプログラムの記法が複雑で静的解析によって誤用箇所を特定できない場合は正しく修正できない．また，プロンプトに十分な REST API 仕様を含めることが困難であるため，LLM が誤用箇所の修正に必要な REST API 仕様を学習していない場合は正しく修正できない．

2.3 RAG

既存の自動バグ修正手法を踏まえて，本研究では REST API の誤用を対象とした RAG (Retrieval-Augmented Generation) の適用による自動バグ修正の有効性を調査する．加藤らは，RAG はプロンプトに与えるテキスト片を検索により決定するアプローチであり，近年の LLM 活用において広く普及している技術であると述べている [20]．入力に関連するテキスト片をプロンプトに含めることで，LLM が

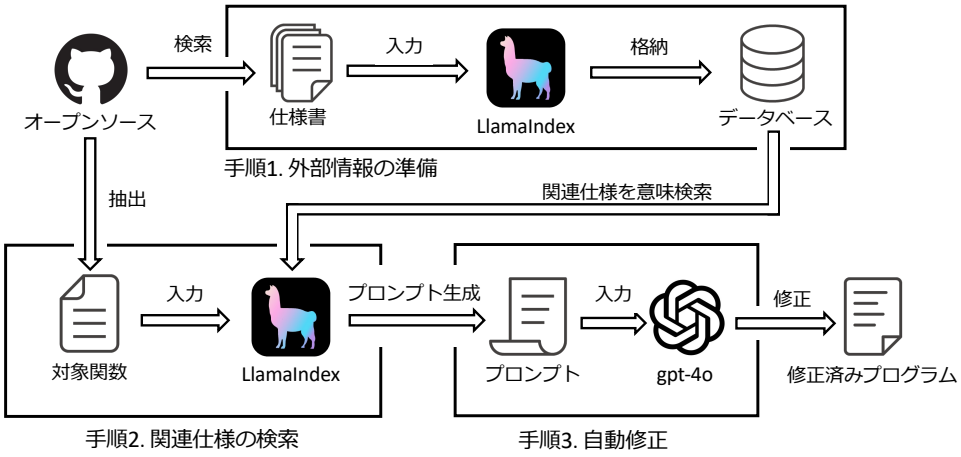

図 2: 標準的な RAG 手法の概要

表 2: プロンプトテンプレート

種類	LLM	RAG	DB 拡張 RAG
基本的な指示	✓	✓	✓
対象関数	✓	✓	✓
仕様リンク	✓		
仕様の一部		✓	✓
仕様の分類			✓

学習していない情報に対しても正しい回答を得ることが期待できる．

3. 調査手法

本章では，評価のベースラインとなる単純な LLM の利用手法について述べる．その後，比較対象として標準的な RAG 手法と，データベースを拡張した RAG 手法について述べる．調査に用いた 3 手法のプロンプトテンプレートの種類は**表 2** に示す．

3.1 単純な LLM の利用

ベースラインとなる本手法は，**表 2** より修正を指示する命令文，非推奨仕様に従った対象関数，REST API プロバイダが公開している REST API 仕様のすべてのリンクをプロンプトに含めて LLM へ入力するアプローチである．本研究では，ChatGPT の gpt-4o を用いて調査を行った．現在，ChatGPT にはリアルタイム検索機能が搭載されている．これを用いることで，ChatGPT は修正に必要な情報をウェブから取得することが可能となる．特に，ChatGPT 自身がウェブ検索を行う必要があると判断した場合は，REST API 仕様のリンクを経由して修正に必要な情報を取得することができる．それにより，修正率の向上が期待できる．

3.2 標準的な RAG 手法

3.1 節で述べたベースライン手法では，REST API 仕様へのリンクをプロンプトに含めることで，幅広い情報へアクセスすることができる．しかし，LLM が修正に必要な REST API 仕様へのリンクを判別できない場合は，関連性が低い REST API 仕様情報の抽出や信頼性が低い技術ブログなどの情報源へアクセスしてしまう可能性があり，修正率低下の原因となる．

そこで，前述した RAG というアプローチを利用した．プロンプトに含めた内容は，**表 2** より修正を指示する命令文や非推奨仕様に従った対象関数に加えて，REST API 仕様の一部を含めて LLM へ入力するアプローチである．**図 2** に本研究で調査する標準的な RAG 手法の概要を示す．標準的な RAG 手法は，以下の手順で構築する．

手順 1. 外部情報の準備

まず，外部情報を準備する必要がある．GitHub やウェブで公開されているオープンソースから対象となる REST API 仕様のリンクを収集する．そのリンクを基に，スクレイピングすることによって仕様情報に関するテキストデータを取得する．次に，テキストデータをベクトル化するプロセスを実行する．ベクトル化とは，テキストデータを数値表現に変換することであり，対象関数に関連する情報を意味的に検索することが可能になる．まず，取得したテキストデータをチャンクと呼ばれる小さなテキスト片に分割する．これにより，それぞれが独立して意味を持つようになり，検索時に正確な関連性を持った情報を取得しやすくなる．続いて，埋め込みモデルを用いて分割したテキスト片にベクトル化処理を行う．埋め込みモデルは，OpenAI 社が提供している text-embedding-ada-002 を使用した．本手法では，テキストデータの分割と埋め込みモデルによるベクトル化処理を LlamaIndex で行った．最後

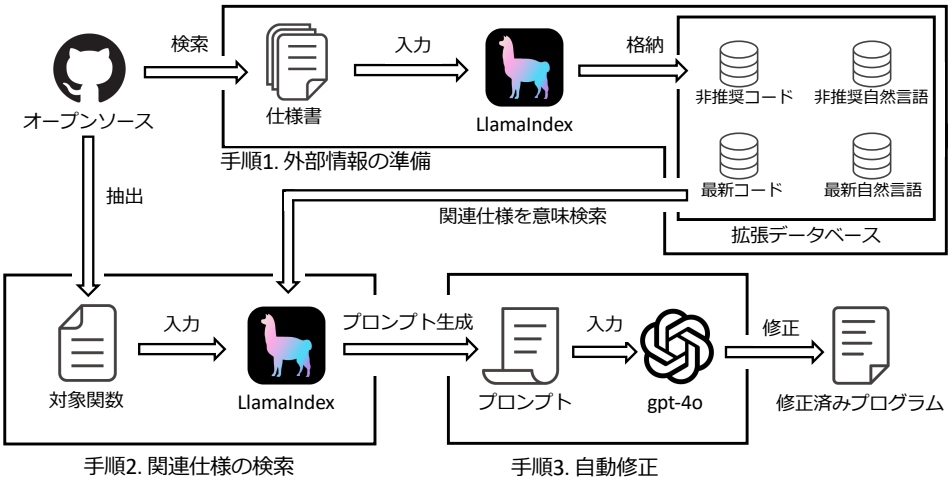

図 3: データベース拡張 RAG 手法の概要

に，ベクトル化を行ったテキストデータをデータベースに格納する．

手順 2. 関連仕様の検索

次に, バグを含むコードに関連する仕様を先程のデータベースから検索する．はじめに, GitHub から REST API 誤用を含む対象関数を抽出する．対象関数は GitHub API を用いて SwitchBot API と Fitbit API の API 誤用を含むものを抽出した．詳細は 3.4 節で記述する．また，対象関数のベクトル化処理を行う．このプロセスにより，対象関数とデータベースに格納した REST API 仕様のテキスト片との意味検索をすることが可能になる．続いて，ベクトル化した対象関数を基にデータベースに対して意味検索を行い，関連するテキスト片を抽出する．テキスト片の抽出数は，上位 20，40，60，80 個でそれぞれ調査して最も修正率が高い結果を採用した．

手順 3. 自動修正

最後に，抽出した結果を基に自動修正を行う．手順 2 で抽出した結果を基に，LLM へ入力するプロンプトを作成する．このプロンプトは，プロンプトエンジニアリングガイド [21] を参考に設計した．詳細は GitHub リポジトリ[*3]を参照されたい．最初に，LLM へ背景情報や実施すべき内容について指示する．その後，データベースから抽出した情報と対象関数を記述し，コンテキストを基に修正を加えるよう指示する．続いて，LLM が生成する回答の形式を記述し，どのように修正して出力すべきか指示する．本手法では，作成したプロンプトを OpenAI 社が提供している gpt-4o へ入力することで回答を生成し，評価を行った．

[*3] https://github.com/shonsukee/APR-prompt

3.3 データベース拡張 RAG 手法

3.2 節で述べたように，RAG を利用することで対象関数と意味的に関連するテキスト片を抽出することができる．しかし，図 2 に示す調査手法では非推奨の REST API 仕様と最新の REST API 仕様を 1 つのデータベースに格納しているため，修正に必要な仕様を十分に抽出できない可能性がある．そこで，以下のようにデータベースを分割する．

- 非推奨の REST API 仕様に関するコード
- 非推奨の REST API 仕様に関する自然言語
- 最新の REST API 仕様に関するコード
- 最新の REST API 仕様に関する自然言語

図 3 に 3.2 節で述べた標準的な RAG 手法を改良した構成の概要を示す．これにより，修正に必要なテキスト片を確実に収集することができ，修正率の向上が期待できる．

実装において標準的な RAG 手法との違いは，データベースへの格納方法やテキスト片の抽出，そしてプロンプトの設計である．手順 1 では，REST API 仕様のテキスト全体を，自然言語とコードに分割する．具体的には，HTML タグを指定することでコードを抽出できる．また，テキスト全体からコード部分を削除することで自然言語を抽出できる．これらを LlamaIndex でベクトル化処理を行い，拡張データベース内の適当なデータベースへ格納する．

手順 2 で関連仕様を意味検索する際は，4 つのデータベースに対してそれぞれ検索する．このとき，3.2 節と同様に対象関数と意味的に関連するテキスト片の上位 20 個をそれぞれのデータベースから抽出する．したがって，最大 80 個のテキスト片をプロンプトに含める．

手順 3 では，プロンプトを改良した．**表 2** より，修正を指示する命令文，非推奨仕様に従った対象関数，REST API 仕様のテキスト片を非推奨仕様と最新仕様で分類して記述した．修正を指示する命令文では，3 点の分析過程を設定

表 3: データセットの誤用種類と件数

誤用の種類	SwitchBot	Fitbit
EP	0	28
QP&EP	0	4
RH&EP	8	0
計	8	32

表 4: データセットの誤用箇所と修正成功数

誤用箇所	LLM	RAG	DB 拡張 RAG
EP	19.2/28	21.4/28	22.2/28
QP & EP	3.0/4	3.4/4	3.0/4
RH & EP	2.0/8	4.6/8	6.8/8
計	24.2/40	29.4/40	32.0/40

した．まず初めに，非推奨仕様を記述して修正すべきプログラムの分析を指示する．次に，対象関数を記述して非推奨仕様に基づいたプログラム誤用箇所の調査を指示する．最後に，最新仕様を記述して対象関数における誤用箇所の修正を指示する．

3.4 評価用データセットの作成

評価のために用いるデータセットを，**表 3** に示す．エンドポイントは EP，クエリパラメータは QP，リクエストヘッダは RH という命名規則に従って記述している．これらのデータセットは，我々が GitHub API によって収集したデータセットである．SwitchBot API もしくは Fitbit API のどちらかを使用しているリポジトリから，すでにクローズされたコミット，イシュー，プルリクエストを収集した．収集手順を以下に示す．

手順 1. GitHub API にクエリを入力してリンクを収集
まず，GitHub API にクエリを入力して，ヒットしたリンクを収集する．その中から，基準に該当するものをプログラムと目視によって抽出した．

手順 2. 基準に該当するものを抽出
コミットとプルリクエストは，修正後に最新の REST API 仕様へ変更されたものを基準とした．イシューは，コメント内に非推奨の REST API 仕様に基づいたプログラムが含まれているもの，もしくはすでにクローズされたコミットおよびプルリクエストのメンションがあるものを基準とした．これらの中で，非推奨の REST API 仕様に基づいたプログラムを目視で抽出して，データセットとした．また，このデータセットを準備するための作業は全て第一著者が行った．

3.5 評価方法

LLM は実行回数によって修正の揺れが生じる可能性がある．それを防ぐために下記の数式を用いることで，データセットを評価した．

$$\text{修正成功数} = \frac{\text{パッチ生成の成功回数}}{5}$$

データセット 1 件あたり 5 回ずつ実行し，パッチ生成の成功回数を，実行回数である 5 で割ることで修正成功数を算出する．また，既存研究 [22] や既存研究で用いられた GitHub リポジトリ[*4]を参考にして，既存研究で提案されているルールのどれかに該当するものを成功と評価した．また，最新の REST API 仕様に基づくプログラムへ修正できているか確認するために，コミット後のソースコードを参照して，最新の REST API 仕様と LLM からの回答を目視で比較した．

調査実験で使用したデータセットやプロンプト，それらを適用した結果は GitHub 上で公開している．[*5] 詳細な結果についてはリポジトリを参照されたい．

本研究の調査実験では，REST API 誤用に対する RAG の有効性やプロンプトの最適化による修正率の変化を検証するために，以下 2 つの RQ を設定する．

RQ1: 単純な LLM の利用と比べて，標準的な RAG 手法の修正率は高いか？
単純な LLM の利用を試みる場合と標準的な RAG 手法を比較し，RAG の有効性を調査する．

RQ2: RAG で用いるデータベースの種類を増やすことで，修正率は向上するか？
RAG で用いるデータベースを拡張し，テキスト片の詳細をプロンプトに記述することで修正率が向上するかを調査する．

4. 調査結果

2 つの REST API 誤用を含むデータセットにおいて，これらの手法を適用して評価を行った．調査結果は 3.4 節と同様の命名規則に従って**表 4** に示す．

4.1 RQ1 の結果

RQ1 について，単純な LLM の利用における平均修正成功回数は次のような結果となった．エンドポイントの誤用は 1 件あたり平均 3.4 回，クエリパラメータとエンドポイントの誤用は 1 件あたり平均 3.8 回，リクエストヘッダとエンドポイントの誤用は 1 件あたり平均 1.3 回，合計で平均 3.0 回修正に成功した．この手法の修正率は 60.5%である．また，40 件中 5 件のデータセットに対しては 1 度も修正できなかった．

それに比べて標準的な RAG 手法における平均修正成功回数は次のような結果となった．エンドポイントの誤用は 1 件あたり平均 3.8 回，クエリパラメータとエンドポイントの誤用は 1 件あたり平均 4.3 回，リクエストヘッダとエンドポイントの誤用は 1 件あたり平均 2.9 回，合計で平均

[*4] https://github.com/SerVal-DTF/APR-Efficiency

[*5] https://github.com/shonsukee/APR-Resources

3.8 回修正に成功した．この手法の修正率は 73.5%である．また，40 件中 4 件のデータセットに対しては 1 度も修正することができなかった．

> RQ1 への回答：単純な LLM の利用と比べて，標準的な RAG 手法の修正率は 13.0%高くなった．

4.2 RQ2 の結果

RQ2 について，前述した標準的な RAG 手法による修正率は 73.5%である．それに比べてデータベースを拡張した RAG 手法における平均修正成功回数は次のような結果となった．エンドポイントの誤用は 1 件あたり平均 4.1 回，クエリパラメータとエンドポイントの誤用は 1 件あたり平均 3.8 回，リクエストヘッダとエンドポイントの誤用は 1 件あたり平均 4.3 回，合計で平均 4.1 回修正に成功した．この手法の修正率は 80.0%である．また，成功回数にばらつきはあるものの，全てのデータセットで少なくとも 1 件以上の修正に成功した．

> RQ2 への回答：RAG で用いるデータベースの種類を増やすことで，修正率が 6.5%高くなった．

4.3 考察

RQ1 について，標準的な RAG 手法による対象関数に関連した REST API 仕様のテキスト片をプロンプトに含めることで，修正率を向上させることができた．調査結果より，単純な LLM の利用ではエンドポイント単体の誤用に対する修正成功件数は多いものの，複数の誤用を含むプログラムに対する修正成功件数は少ないことが明らかとなった．調査に使用した gpt-4o は学習データが 2023 年 10 月時点であるが，対象とした SwitchBot API や Fitbit API はそれ以前に行われた仕様変更であるため，REST API 仕様を十分学習していると考えられる．さらに，ウェブ検索機能によって，最新の仕様情報にアクセスすることができる．しかし，対象関数の REST API 誤用箇所を特定できず，誤用箇所を修正できなかったことが修正率低下の原因であると考えられる．また，プロンプトに含めたリンク数やリンク先の情報量が多く，必要な情報を判別することが困難であったことも修正率低下の原因として挙げられる．

標準的な RAG 手法では，LLM によるウェブ検索を必要とせず，修正時に参照すべき情報をプロンプトに含めることができるため，修正率の向上に寄与したと考えられる．対象関数の誤用箇所を明示的に指定しなくとも，REST API 仕様として提供する情報量を適切に制限することで，正確な修正を行うことができたと考えられる．さらなる修正率向上のための課題として，非推奨と最新の REST API 仕様を分類してプロンプトへ挿入することが挙げられる．LLM が非推奨仕様に基づいて修正を行う可能性を排除するため，非推奨と最新の REST API 仕様を分類して記述することが修正率の向上につながると考えられる．

RQ2 について，RAG で用いるデータベースの種類を増やすことで修正率を向上させることができた．データベースを拡張した RAG 手法では，非推奨と最新の REST API 仕様を分類してプロンプトに挿入したことが修正率の向上に寄与したと考えられる．具体的には，LLM が非推奨仕様を参照することで対象関数の誤用箇所を特定し，最新仕様を参照しながらその誤用箇所を適切に修正することが可能になった．また，それぞれの REST API 仕様のテキスト片を分類して入力することで，標準的な RAG 手法では修正できなかったデータセットを修正することに成功したケースもあった．そのため，プロンプトに含めるテキスト片についての説明を詳細に記述することが，修正率の向上につながったと考えられる．

5. 関連研究

これまで，Java の API を対象とした誤用検出 [8], [9] や GPT-3[10] などの事前訓練済みの大規模言語モデル (LLM) を活用したソースコードの自動修正 [11], [12], [13], [14] に関する研究が行われてきた．Li らは，クライアントコード，API 仕様書，ライブラリコードから制約を抽出して Java の API 制約グラフを構築することで誤用を検出するアプローチを提案している [8]．Ren らは，Java API の制約に関係する知識グラフを構築し，それを基に誤用を検出している [9]．Xia らは，既存の自動プログラム修正ツールと比較して，LLM の方が多くの自動修正を行う能力を有していることを示した [11]．Kechagia らは，Java の API 誤用からなるベンチマークを用いて既存の APR ツール 14 個を自動実行し，実行時間と修正率を評価している [12]．荒木らは，Java の API 誤用を含むプログラムと，それに類似する API 利用パターンを比較することで不足しているメソッド呼び出しを追加する自動修正アプローチを提案している [13]．Yin らは，Chain-of-Thought (CoT) による収集フェーズと，few-shot と CoT による修正フェーズを組み合わせた自己指示型 LLM ベースの自動修正手法を提案し，Java の API 誤用に適用することで有効性を示している [14]．

また，RAG を用いた自動バグ修正の研究が行われている [23]．Wang らは，Java の API 誤用を含むプログラムに対して，語彙的および意味的に関連性の高いバグと修正例のペアを検索し，それを用いて自動修正する手法を提案している [23]．本研究では，REST API の仕様書を検索して自動修正する調査を行っている．

6. まとめ

本研究では，REST API 誤用を含むプログラムを対象に，LLM による修正率の向上を目的として，複数のプロンプト

作成アプローチによる性能を調査した．標準的な RAG 手法では，プロンプトに修正の命令文や対象関数に加えて，関連する REST API 仕様のテキスト片を含めることで修正率の向上が確認できた．データベースを拡張した RAG 手法では，REST API 仕様のテキスト片を非推奨と最新で分類することにより，さらなる修正率の向上が確認できた．

今後の課題として，プロンプトに挿入する REST API 仕様のテキスト片をコードと自然言語で分割して渡すことにより，対応関係が失われる問題がある．そのため，データベースから自然言語とコードのセットを取得するような仕組みを構築することで，さらなる修正率向上に寄与すると考えている．また，本研究で用いたデータセット数が少ないため，より多くの事例を用いた検証をすることで RAG の汎用的な有効性を示したいと考えている．

謝辞 本研究を進めるに当たり，有益な助言をいただいた，島根大学 神谷年洋教授，立命館大学 西本優作氏に感謝いたします．本研究は，JST さきがけ JPMJPR21PA ならびに JSPS 科研費 JP20K11745，JP24K02923 の支援を受けたものです．

参考文献

[1] Makhshari, A. and Mesbah, A.: IoT bugs and development challenges, *in Proc. of ICSE*, IEEE, pp. 460–472 (2021).
[2] Schwandt, F.: Internet of things (IoT) connected devices installed base worldwide from 2015 to 2025 (in billions), *Statista* (2016).
[3] 大野堅太郎, 吉田則裕, 朱　文青, 高田広章：コードクローン検出に基づく IoT を対象とした自動パッチ生成，第 29 回ソフトウェア工学の基礎ワークショップ (FOSE2022), pp. 171–180 (2022).
[4] 山岸克紀，吉田則裕，槇原絵里奈：REST API 仕様に基づく大規模言語モデルを用いた自動バグ修正手法，ソフトウェアエンジニアリングシンポジウム 2024 論文集，pp. 155–164 (2024).
[5] Fielding, R. T.: Architectural styles and the design of network-based software architectures, PhD Thesis, University of California, Irvine (2000).
[6] Bodenheim, R., Butts, J., Dunlap, S. and Mullins, B.: Evaluation of the ability of the Shodan search engine to identify Internet-facing industrial control devices, *IJCIP*, Vol. 7, No. 2, pp. 114–123 (2014).
[7] Jiang, X., Lora, M. and Chattopadhyay, S.: An experimental analysis of security vulnerabilities in industrial IoT devices, *ACM TOIT*, Vol. 20, No. 2, pp. 1–24 (2020).
[8] Li, C., Zhang, J., Tang, Y., Li, Z. and Sun, T.: Boosting API Misuse Detection via Integrating API Constraints from Multiple Sources, *in Proc. MSR*, pp. 14–26 (2024).
[9] Ren, X., Ye, X., Xing, Z., Xia, X., Xu, X., Zhu, L. and Sun, J.: API-misuse detection driven by fine-grained API-constraint knowledge graph, *in Proc. of ASE*, pp. 461–472 (2020).
[10] Brown, T. B., Mann, B., Ryder, N., Subbiah, M., Kaplan, J., Dhariwal, P., Neelakantan, A., Shyam, P., Sastry, G., Askell, A., Agarwal, S., Herbert-Voss, A., Krueger, G., Henighan, T., Child, R., Ramesh, A., Ziegler, D. M., Wu, J., Winter, C., Hesse, C., Chen, M., Sigler, E., Litwin, M., Gray, S., Chess, B., Clark, J., Berner, C., McCandlish, S., Radford, A., Sutskever, I. and Amodei, D.: Language models are few-shot learners, *in Proc. of NeurIPS*, pp. 1877–1901 (2020).
[11] Xia, C. S., Wei, Y. and Zhang, L.: Automated program repair in the era of large pre-trained language models, *in Proc. of ICSE*, pp. 1482–1494 (2023).
[12] Kechagia, M., Mechtaev, S., Sarro, F. and Harman, M.: Evaluating automatic program repair capabilities to repair API misuses, *IEEE TSE*, Vol. 48, No. 7, pp. 2658–2679 (2021).
[13] 荒木良仁，桑原寛明，國枝義敏：API 利用パターンを用いた自動プログラム修正手法，情報処理学会研究報告，Vol. 2021-SE-207, No. 3, pp. 1–9 (2021).
[14] Yin, X., Ni, C., Wang, S., Li, Z., Zeng, L. and Yang, X.: ThinkRepair: Self-directed automated program repair, *in Proc. of ISSTA*, pp. 1274–1286 (2024).
[15] Lewis, P., Perez, E., Piktus, A., Petroni, F., Karpukhin, V., Goyal, N., Küttler, H., Lewis, M., Yih, W.-t., Rocktäschel, T., Riedel, S. and Kiela, D.: Retrieval-augmented generation for knowledge-intensive NLP tasks, *in Proc. of NeurIPS*, pp. 9459–9474 (2020).
[16] SwitchBot: SwitchBot API v1.1, (online), available from ⟨`https://github.com/OpenWonderLabs/SwitchBotAPI`⟩ (accessed 2025-01-25).
[17] Fitbit: Fitbit API, (online), available from ⟨`https://dev.fitbit.com`⟩ (accessed 2025-01-25).
[18] Liu, K., Koyuncu, A., Kim, D. and Bissyandé, T. F.: TBar: Revisiting template-based automated program repair, *in Proc. of ISSTA*, pp. 31–42 (2019).
[19] Li, G., Zhi, C., Chen, J., Han, J. and Deng, S.: Exploring Parameter-Efficient Fine-Tuning of Large Language Model on Automated Program Repair, *in Proc. of ASE*, pp. 719–731 (2024).
[20] 加藤　整，原田智彦，下條ひなた，杉浦健一，竹之内啓太：大規模言語モデルを用いた広範なソースコード理解とドキュメント生成手法の検証，ソフトウェアエンジニアリングシンポジウム 2024 論文集，pp. 137–144 (2024).
[21] DAIR.AI: Prompt Engineering Guide, (online), available from ⟨`https://www.promptingguide.ai`⟩ (accessed 2025-01-25).
[22] Liu, K., Wang, S., Koyuncu, A., Kim, K., Bissyandé, T. F., Kim, D., Wu, P., Klein, J., Mao, X. and Traon, Y. L.: On the efficiency of test suite based program repair: A systematic assessment of 16 automated repair systems for java programs, *in Proc. of ICSE*, pp. 615–627 (2020).
[23] Wang, W., Wang, Y., Joty, S. and Hoi, S. C.: RAP-Gen: Retrieval-augmented patch generation with CodeT5 for automatic program repair, *in Proc. of ESEC/FSE*, pp. 146–158 (2023).