一般社団法人　電子情報通信学会
THE INSTITUTE OF ELECTRONICS,
INFORMATION AND COMMUNICATION ENGINEERS

信学技報
IEICE Technical Report

ソースコード問い合わせのための長コンテキストLLM向け RAG手法の提案

神谷　年洋†

† 島根大学学術研究院理工学系　〒690–8504 島根県松江市西川津町 1060
E-mail: †kamiya@cis.shimane-u.ac.jp

あらまし　大規模言語モデル (LLM) のコンテキスト長制限は緩和されてきているとはいえ，現状ではソフトウェア開発タスクへの適用を困難にしている．本研究では，ソースコードに関する問い合わせのために，実行トレースを RAG に取り入れた手法を提案する．小規模な実験により，手法が LLM 応答品質に寄与する傾向を確認した．
キーワード　RAG, LLM, ソフトウェア開発

A RAG Method for Source Code Inquiry Tailored to Long-Context LLMs

Toshihiro KAMIYA†

† Institute of Science and Engineering, Shimane University 1060 Nishikawatsu-cho, Matsue, Shimane, 690–8504 Japan
E-mail: †kamiya@cis.shimane-u.ac.jp

Abstract　Although the context length limitation of large language models (LLMs) has been mitigated, it still hinders their application to software development tasks. This study proposes a method incorporating execution traces into RAG for inquiries about source code. Small-scale experiments confirm a tendency for the method to contribute to improving LLM response quality.
Key words　RAG, LLM, Software Development

1. はじめに

テキスト生成 AI（大規模言語モデル; LLM）の発展が著しい．ソフトウェア開発においても LLM が利用されるようになり，GitHub Copilot などの AI アシスタントツールが登場している．しかし，LLM には課題も存在する．

LLM の問題としてよく指摘されているものに，コンテキスト長制限がある．コンテキスト長制限とは，LLM がテキストを生成する際に考慮できるコンテキストの長さの上限である．例えば，OpenAI 社の `gpt-4-32k` という LLM ではコンテキスト長制限は 3 万 2 千トークン（英語では 6 万文字程度）である．ChatGPT のようなチャット形式の UI から LLM を利用する際には，入力テキストがこの制限を超えるとエラーになるような実装が一般的である．長コンテキスト入力をサポートする LLM が登場してきているが，そのような LLM であっても，Levy らの研究 [2] によれば，入力テキストが長くなるほど LLM の推論性能が低下する，いわゆる**干し草の中の針** (needle in a haystack) 問題に突き当たる．

ソフトウェアのソースコードは簡単に数万行以上のサイズになるため，ソースコードを含めた問い合わせがコンテキスト長制限を超えることがあり，品質の高い回答を得ることが困難になる．現在のソフトウェア開発では大規模な再利用が行われる．開発されているプロダクト自体のソースコードは短くても，再利用されているライブラリ（フレームワークも含めて）のソースコードを含めると数十倍になることが一般的である．そのため，LLM のコンテキスト長制限は，ソフトウェアの不具合の原因の特定や，特定の機能の実装を調べるといった問い合わせの際に問題となる．

LLM のコンテキスト長制限を緩和するための方法の一つに RAG（Retrieval-Augmented Generation; 検索拡張生成）[3] がある．RAG では，プロンプトで問い合わせている内容に関係のあるドキュメントを，データベースやウェブ検索など，何らかの方法により検索，選別し，そのようなドキュメントを元のプロンプトに付加して LLM に入力することで，ドキュメントの内容に即した回答を得る．例えば，プロンプトが「猫の習性について教えてください」の場合，Wikipedia の「Cat」の記事を検索し，そのテキストをプロンプトに付加することで，より詳細な回答を得ることができる．ソフトウェアプロダクトに対する問い合わせであれば，RAG により元のプロンプトに付加するドキュメントとして，そのプロダクト自身や再利用されているプロダクトのソースコードを用いることができる．

LLM のコンテキスト長制限を緩和するための他の方法とし

ては，はじめから大きなコンテキストを持つ LLM を利用する方法がある．ただし，現在広く用いられている Transformer 系列の LLM の場合，コンテキストの長さの 2 乗に比例した計算コストが必要となるため，コンテキスト長を大きくすることは即，計算コストの増大につながる．

本研究では，ソフトウェアプロダクトのソースコードに関する問い合わせを目的とした RAG の手法を提案する．提案手法では，プロダクトを実行して実行トレース（呼び出された関数のログ）を取得し，その実行トレースからコールツリーや呼び出された関数のソースコードを抽出する．それらを RAG のドキュメントとして LLM に入力し，ソースコード全体を参照させずに的確な回答を得ることで，干し草の中の針問題を緩和・解決することを狙う．

2. 関 連 研 究

ソフトウェア開発に LLM を応用するための研究として，LLM をバグ位置特定に利用する研究 [1] では，失敗したテストケースのエラーメッセージなどを LLM に入力し，原因を特定させ，さらに，バグの位置を特定させるという手法が提案されている．

開発者がソースコードや API を理解する作業を支援するために IDE に LLM を組み込んだツールが提案されている [4]．開発者の質問に答えたり，自動的にコードの要約を表示する機能を提供するなど，開発者の集中を途切れさせずに作業の効率化を図ることが目的とされている．

文献 [6] では，訓練データに解答例が含まれているために既存のベンチマークが LLM の性能を正しく評価できていない可能性があることを指摘している．後述する 4.4 節の実験でも一部問題となっている．

ソフトウェア開発 AI アシスタントツールである GitHub Copilot のブログ[1]には，AI から適切な回答を得るためには，現在取り組んでいるタスクに関係のあるファイルを開くことで，LLM に適切なコンテキストを与えるようにせよとの記述がある．本研究で提案する手法は，対象プロダクトに動的解析を行い LLM に見せるべきソースコードを自動的に特定するものである．

3. アプローチ

提案手法は，ユーザーからの問い合わせとソフトウェアの実行トレースを入力とし，以下のステップで対応するソースコードを特定し，LLM に入力する．

ステップ 1. ユーザーは，ソフトウェアプロダクトに対する問い合わせと，その問い合わせに関連する機能の実行時に収集したトレース (呼び出された関数のログ) を入力する．実行トレースの取得は，著者が開発している実行トレース取得ツール `rapt` により行う．このツールは対象の Python スクリプトを実行し，呼び出された関数のログを収集する．この際，関数のソースコード上の位置情報（ソースファイルや行番号）も収集される．例えば，4.4 節で後述する実験では，実験対象として CLI（コマンドラインインターフェイス）ツールである `rich-cli` の CSV ファイルを整形して表示する機能に関する問い合わせをしているが，その際には CSV ファイルを表示させて実行トレースを取得している．

ステップ 2. 実行トレースを解析し，実行された関数やメソッド，それらの呼び出し関係を特定する．

ステップ 3. 呼び出された関数名から，対応するソースコードファイルとその中の関数の位置を特定する．

ステップ 4. 呼び出し関係からコールグラフ (関数の呼び出し関係を表すグラフ) を作成し，さらにループなどを取り除いてコールツリー (木構造) を生成する．

ステップ 5. 問い合わせのテキストに，コールツリーと，コールツリー内の関数のソースコードを付加したテキストをプロンプトとして LLM に入力し，応答を出力する．

この際，関数のソースコードはコールツリー内で出現する順になるようにした．すなわち，ある関数 `f` が別の関数 `g` を呼び出していれば，`f` のソースコードの次に `g` のソースコードが並べられる．この順序付けにより，LLM にソースコードの各行を実行順に示すことで，実行のフローを追いやすくする狙いがある．後述の 4.4 節の実験では，このような順序付けの効果を評価している．

3.1 実用上の工夫

提案手法の実装では，コーツルリーをコンパクトな表現とするために，次の処理を行う．

(1) ある関数が同じ関数を複数回呼び出すときには，呼び出されている関数を 1 つのノードで表現する．また，ある関数が異なる関数から呼び出されているときには，呼び出されている関数のノードは別のものとして表現する（マージしない）．

結果として，このようなコールツリーは，有向グラフとして表現したコールグラフを，再帰呼出し（すなわちグラフ内の循環）や合流（すなわち，グラフ内ノードに複数の入り辺がある部分）を取り除いた木になる．木構造にすることで，プロンプトにソースコードを付加する際の順序を判別しやすくなると期待される．

(2) 再帰呼出しがあるときには，再帰的な呼び出しとなった最初のノードが呼び出される階層までが表現され，それより深い呼び出しのノードは省略される．再帰呼出しは処理の繰り返しのために利用されることがあり，そのような場合に非常に大きなコールツリーとなることを避ける．

(3) その他の工夫と制限

実行トレース取得ツール `rapt` は，ユーザーが指定したモジュールに含まれる関数の呼び出しのみを記録する．組み込み関数（`print` など）や標準ライブラリに属する関数の呼び出しは記録されない．

`rapt` では，対象プロダクトの実行中に関数をラップして呼び出し時にログを記録する方法を採用している．しかし，この実装方式の制限により，動的に（遅延して）ロードされるモジュールの関数や，グローバルスコープにはない関数（ラムダなど）の呼び出しは記録することができない．

（注1）：Using GitHub Copilot in your IDE: Tips, tricks, and best practices, `https://github.blog/2024-03-25-how-to-use-github-copilot-in-your-ide-tips-tricks-and-best-practices/`

表 1　対象プロダクトを構成するパッケージ

パッケージ	バージョン	Python 行数
certifi	2024.2.2	63
charset-normalizer	3.3.2	4,022
click	8.1.7	5,659
docutils	0.20.1	28,303
idna	3.6	11,142
linkify-it-py	2.0.3	2,032
markdown-it-py	3.0.0	4,226
mdit-py-plugins	0.4.0	2,440
mdurl	0.1.2	342
Pygments	2.17.2	94,240
requests	2.31.0	2,904
rich	13.7.1	19,638
rich-cli	1.8.0	900
rich-rst	1.2.0	569
textual	0.54.0	33,438
typing_extensions	4.10.0	1,633
uc-micro-py	1.0.3	14
urllib3	2.2.1	6,419
(合計)		217,984

一般に，プログラムはスタートアップ時にモジュールのインポートや初期化を行う．このような処理を分析から除外したいことがあるため，コールツリーの枝刈りを行う機能を実装した．枝刈りを行うにはまず，プログラムの機能を実行しない実行トレース（プログラムを開始してすぐに終了させる）を取得する．そのようなコーツルリーに含まれる関数はスタートアップに関するものであるとみなし，分析対象のコールツリーの葉に同じノード（同じ関数呼び出し）があったときには削除する．

4. 実　　験

本節では，オープンソースのプロダクトを対象として適用することで提案手法を評価する．ソフトウェアの開発で起こり得ると想定される具体的な問い合わせと，その問い合わせに対する応答の品質を評価する基準（後述）を用意した．プロンプトの内容（コールツリーや関数のソースコードの有無や順序）を変化させた変種を作り，問い合わせを行って応答の品質を評価（評価基準は後述）することで，提案手法が応答の品質に寄与しているかを分析する．

4.1　対象プロダクト

実験の対象には OSS のマンドラインのツール `rich-cli`[2]を選定した．選定の理由として，機能が豊富であり実験に適切な規模（約 22 万行）を持つことと，CLI ツールであるために繰り返し実行する際に条件を揃えやすい（再現性が高い）ことが挙げられる．ツール `rich-cli` は，CSV や Markdown，ReStructuredText といったフォーマットのファイルを入力し，シンタックスハイライト等の整形を行ってターミナルに出力する機能を持つ．

表 4.5 に対象プロダクト，直接および間接に依存しているライブラリ（パッケージ）について，Python で記述されたソースファイルの行数（ツール `cloc`[3]により計測）を示す．対象プロダクトである `rich-cli` は，ライブラリ `rich` や `rich-rst` の機能をコマンドラインから利用できるようにするツールという位置づけであるため，単体では 900 行であるが，依存するライブラリまで含めると合計約 22 万行となる．

（注2）：rich-cli `https://github.com/Textualize/rich-cli`

表 2　実験に利用した LLM

LLM	c.	利用サービス
Gemini 1.5 Pro	1M	Google AI Studio から利用
Claude 3 Sonnet	200K	poe.com から利用
ChatGPT-4	128K	poe.com から利用

c. の欄は，コンテキスト長制限（トークン数）を示す．

表 3　プロンプトの変種

変種	説明
full	提案手法のプロンプトそのもの．問い合わせのテキスト，コールツリー，コールツリー内の関数のソースコードをコールツリー内で出現した順に含む．コールツリー内で同じ関数が複数回出現する場合には，同じ関数のソースコードが複数回プロンプトに含まれる．
A	full と同じだが，コールツリー内のソースコードは関数名（モジュール名も含む）でソートされたプロンプト．コールツリー内で同じ関数が複数回出現する場合，その関数のソースコードは重複させずに 1 回だけプロンプトに含める．
C	full からコールツリーを取り除いたもの．関数のソースコードがコールツリー内で出現した順に含まれる．
CA	full からコールツリーを取り除いたもの．関数のソースコードはソートされ重複が排除されている．
T	full からソースコードを取り除いたもの．

4.2　実験に使用した LLM

表 2 に実験に使用した LLM である Gemini 1.5 Pro [5], Claude 3 Sonnet, GhatGPT-4 それぞれのコンテキスト長制限（トークン数）を示す．4.5 節で後述するように，生成されるプロンプトの長さは最大で 8 万 7 千トークンほどになるため，比較的大きなコンテキストを扱うことができる LLM を選定した．いずれも API を利用せず，チャット形式の UI から，プロンプトを貼り付けることにより応答を生成している．各 LLM はそれぞれ異なるトークナイザー（テキストをトークンに分割する機能）を使用していて，同じテキストでも異なるトークン数となるため，コンテキスト長制限はあくまで目安である．

4.3　プロンプトの変種

提案手法では，プロンプトにコールツリーやソースコードを含める．これらをプロンプトに含めることによる効果を実験的に評価するため，提案手法のプロンプトからこれらを削除したものや，順序を変更した変種を作成し，応答の品質を評価する．表 3 に実験で用いたプロンプトの変種を示す．図 1 に，次の 4.4 節で利用したプロンプトの例を示す．

4.4　実　験　1

ソフトウェアの開発で起こり得ると想定される具体的な問い合わせについて，提案手法にしたがってプロンプトを作成し，

（注3）：Cloc `https://github.com/AlDanial/cloc`

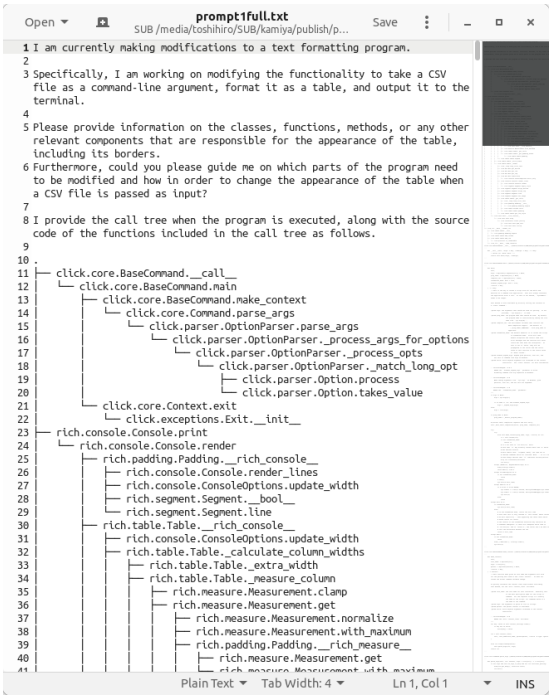

図 1　プロンプトの例

LLM に入力してその応答を評価基準により評価する．本実験では，応答の生成は各プロンプトの各変種，各 LLM について 1 回だけ行っている．チャット形式の UI による LLM のサービスでは乱数を用い，応答を生成するたびに異なる内容となる．その意味でも，応答の評価は絶対的なものではなく，全体的な傾向を見るためのものであることに留意する必要がある．

4.4.1　プロンプト 1

ツールは CSV ファイルを整形してターミナルに表示する機能を持つ．この機能では罫線文字を利用してテーブルの外見を作っている．問い合わせとして，この外見（罫線の種類や右寄せなど）を決めている関数がなんであるか，および，外見を変更する方法を尋ねた．ツールのコマンドラインではフォーマットを変更する方法は提供されていないため，ソースコードの修正が必要となる．修正すべき箇所の正解は，変更によってプロダクトの機能が変更されることを確認でき，かつ，指摘された場所以外の修正が必要ではないこと，プロダクトの他の機能になるべく影響を及ぼさないことを勘案して決定した．

応答のそれぞれを，著者が目視で調査した上で，次の評価基準に照らして 4 点満点で評価した．応答の中で説明が間違っていたりハルシネーションを起こしていると判断される項目は減点した．

① 機能を実現しているクラスや関数の名前を出力できる（1 点）．複数指摘していて正解が含まれる場合は 0.5 点．
② テーブルの外見の要素として罫線以外にも色やパディングなどがあることを説明している（1 点）．
③ 変更の内容（変更後のコードまたは変更の仕方）を出力できる（1 点）．複数指摘していて正解が含まれる場合は 0.5 点．
④ 変更すべき箇所を関数名により出力できる（1 点）．複数指摘していて正解が含まれる場合は 0.5 点．

表 4　プロンプト 1 の応答の評価値

LLM	full	A	C	CA	T
Gemini 1.5 Pro	4.0	4.0	4.0	3.0	4.0*
Claude 3 Sonnet	4.0	4.0	4.0	4.0	4.0
ChatGPT-4	3.0	4.0	4.0	3.0	4.0*

*を付した数字は，コールツリーには含まれないグローバル変数の名前が応答に含まれていたことを示す．

表 5　プロンプト 2 の応答の評価値

LLM	full	A	C	CA	T
Gemini 1.5 Pro	4.0	4.0	4.0	2.0	3.0*
Claude 3 Sonnet	2.0	3.5	1.5	2.5	2.5
ChatGPT-4	4.0	3.0	3.0	3.0	3.5*

*を付した数字は，コールツリーには含まれないグローバル変数の名前が応答に含まれていたことを示す．

プロンプト 1 の結果を表 4 に示す．すべての LLM，すべての変種から得られた応答の評価値が大きな値となった．ただし，変種 T，すなわち，問い合わせのテキストと，関数名がノードとなっているコールツリーのみを含むプロンプトに対する応答に，グローバル変数の名前が含まれるものがあった．ハルシネーション，または，LLM の訓練データに `rich` パッケージに関する情報が含まれていて，プロンプトの内容ではなく学習した内容に基づいて応答を生成した可能性を排除できない（**データ漏洩疑い**）．

4.4.2　プロンプト 2

ツールはテーブルが含まれている Markdown ファイルを整形してターミナルに表示する機能を持つ．プロンプト 1 と同様に見た目を変更する方法を尋ねた．

Markdown にはテーブル以外にも箇条書きや引用など多くの書式があるため，CSV ファイルの処理と比較すると，より複雑な構文解析が行われる．そのため，処理に関係するクラスやメソッドの数も多くなっている点で，プロンプト 1 よりも難易度が高くなっている．評価基準はプロンプト 1 と同様のものを用いた．

プロンプト 2 の結果を表 5 に示す．すべての LLM において，変種 full または A から得られた応答の評価値が最大となった．また，プロンプト 1 の場合と同様，変種 T の応答にデータ漏洩疑いが含まれた．

4.4.3　プロンプト 3

プロンプト 1 で利用した CSV ファイルを整形してターミナルに表示する機能と，プロンプト 2 で利用したテーブルが含まれている Markdown ファイルを整形してターミナルに表示する機能を比較する．具体的には，実装の違いと共通点，制御フローやデータ構造の違い，テーブルに関する機能の違いを尋ねた．プロンプト 3 は実験中最長のものであり，手法のスケーラビリティを試すベンチマークにもなっている．評価基準は次のものを用いた．

① 実装の違いと共通点を説明している（1 点）
② 制御フローの違いを示す重要な関数やメソッドの名前を示している．両者ともに正解で 1 点．一方正解なら 0.5 点．

表 6　プロンプト 3 の応答の評価値

LLM	full	A	C	CA	T
Gemini 1.5 Pro	3.5	4.0	2.0	3.0	1.5
Claude 3 Sonnet	2.0	2.0	1.5	2.0	3.0
ChatGPT-4	1.0	3.5	2.5	1.0	3.0

表 7　プロンプト 4 の応答の評価値

LLM	full	A	C	CA	T
Gemini 1.5 Pro	4.0	2.5	2.0	3.0	1.0
Claude 3 Sonnet	4.0	3.0	2.0	2.0	3.0
ChatGPT-4	2.5	2.5	2.5	2.5	2.5

③ 利用されているデータ構造の違いをクラス名で説明できる．両者ともに正解で 1 点．一方正解なら 0.5 点．
④ テーブルに関する両者の機能の違い（右寄せの有無など）を説明できる（1 点）．Markdown 一般について（リンクなど）は除外する．右寄せなどの具体的な例を出さずに単にスタイルなどと表現している場合は 0.5 点．

プロンプト 3 では 2 つの実行を比較する必要があるため，プロンプトの内容は順に，問い合わせのテキスト，1 回めの実行のコールツリー，1 回めの実行のコールツリーに含まれる関数のソースコードの並び，2 回めの実行のコールツリー，2 回めの実行のコールツリーに含まれる関数のソースコードの並び，とした．ただし，変種 CA のプロンプトの内容は，問い合わせのテキスト，いずれかの実行のコールツリーに含まれる関数のソースコードを関数名でソートしたもの，とした．

プロンプト 3 の結果を表 6 に示す．ChatGPT-4 の変種 full の評価値が小さいことが目立つ．あくまで推測であるが，プロンプト 3 の変種 full はこの実験における最長のプロンプトでもあるため（4.5 節で後述），LLM のコンテキスト長制限に近くなり，応答の品質が低下した可能性がある（**コンテキスト長制限疑い**）．

4.4.4　プロンプト 4

ツールには絵文字のコード（`:sparkle:` など）を絵文字に変換して出力するコマンドラインオプション `--emoji` がある．このオプションはコマンドライン引数でテキストを与えたときは有効であるが，ファイルで与えたときには機能しない．その原因と修正方法を尋ねるものとした．

コマンドライン引数で与えたテキストとファイルとでは異なる実装になっているため，ファイルを処理する処理に絵文字に関する処理を追加する必要がある．オプションの有無による場合分けも必要となる．また，ツールの機能の一貫性を保つためには，既存の処理を再利用する修正が望ましいため，再利用に関しても評価項目に加えた．プロンプト 4 は，必要な処理をプロダクトのどこに追加するべきかという設計に関する判断が必要とされ，これまでのプロンプトと比較して難易度が高いものとなっている．

評価基準は次のものとした．
① 原因を出力できる（1 点）．正解を含めた複数の原因を指摘している場合は 0.5 点．

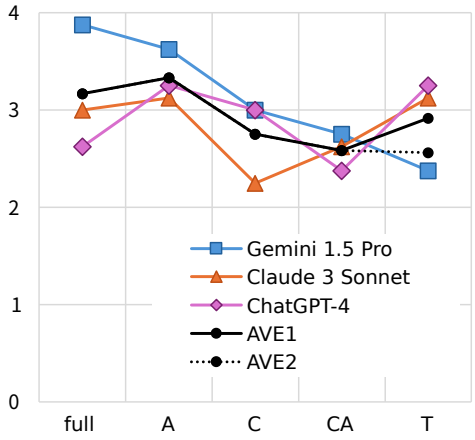

図 2　評価値の傾向

② 変更の内容（変更後のコードまたは変更の仕方）を出力できる（1 点）．複数指摘していて正解が含まれる場合は 0.5 点．
③ 変更すべき箇所を関数名により出力できる（1 点）．複数指摘していて正解が含まれる場合は 0.5 点．
④ 既存の機能を再利用した修正プランを出力できる（1 点）．新規に機能を作り込むプランの場合は 0.5 点．

プロンプト 4 の結果を表 7 に示す．すべての LLM で変種 full の評価値が最大になった（ただし，ChatGPT-4 ではすべての変種で評価値が 2.5 であった）．

4.4.5　分　　析

図 2 に，各変種（full, A, C, CA, T）のプロンプト 1 から 4 の平均を折れ線グラフとして，横軸にプロンプトの変種，縦軸に評価値を集計した値を示す．黒い実線（AVE1）が実験結果の値そのままを用いたもの，黒い点線（AVE2）がプロンプト 1 および 2 で観察されたデータ漏洩疑いを除いたものである．例えば，左端にある黒丸は，変種 full について，すべての LLM，すべてのプロンプト（1 から 4 まで）の評価の平均値が 3.17 であることを示す．

黒い実線以外の折れ線は，LLM ごとに分類して集計したものを示す．full のところで評価値が下がっている部分は，実験 1 のプロンプト 3 で観察されたように，コンテキスト長制限が影響した可能性がある．

本実験はサンプルサイズが小さいため，統計的な判断ができないながらも，全体的な傾向について述べる．

まず，コンテキストが最も大きい LLM である Gemini 1.5 Pro について見れば，変種 full の評価値が最大であり，あとは A，C，CA，T の順となっていた．

ソースコードの有無に着目すると，ソースコードを含まない変種 T の評価値は，ソースコードを含み，かつ，コールツリーを含む変種である full や A の評価値に及ばない．このことから，コールツリーとソースコードをプロンプトに含めることは，応答の品質に寄与するという傾向が読み取れる．

ソースコードを含む変種，すなわち，full, A, C, CA に着目すると，関数が呼び出される順序を含まない変種は CA のみである．それ以外の変種 full, A, C は，コールツリー，または，関数

表 8　プロンプトのサイズ（ChatGPT-4 のトークン数）

プロンプト	full	A	C	CA	T
プロンプト 1	32,250	19,949	22,079	18,768	1,279
プロンプト 2	64,838	53,537	61,238	49,943	3,711
プロンプト 3	87,950	73,338	83,198	50,528	4,876
プロンプト 4	26,104	23,984	24,489	18,792	1,751

表 9　プロンプトが参照する Python ソースコード

プロンプト	ファイル数	ファイル行数	full 行数
プロンプト 1	14	11,552	2,549
プロンプト 2	53	18,799	7,079
プロンプト 3	53	18,799	9,626
プロンプト 4	18	13,757	2,737

プロンプト 2 と 3 は同じソースファイルの集合を参照しているため，ファイル数やファイル行数も同じ値となっている．

のソースコードの並び順により，関数が呼び出される順序の情報がプロンプトに含まれている．変種 CA の評価値が小さいことにより，関数が呼び出される順序は応答の品質に寄与するという傾向が読み取れる．

さらに，変種 full, A, C に着目すると，変種 C のみソースコードの並び順により関数が呼び出される順序が提示され，full と A はコールツリーにより関数が呼び出される順序が提示されている．実験 1 のプロンプト 3 で述べたコンテキスト長制限疑いを除外すれば，コールツリーによる提示は応答の品質に寄与する傾向が読み取れる．

4.5 実　験　2

提案手法により作成されるプロンプトのサイズを評価する．

(1) LLM のコンテキスト長制限に対してプロンプトのサイズがどの程度の大きさであるかを調べる．

表 8 に，プロンプト 1 から 4 のプロンプト長を ChatGPT-4 のトークナイザーで計測したものを示す[4]．最大のものはプロンプト 3 の変種 full で 87,950 トークンである．計算上は，今回利用した ChatGPT-4 のコンテキスト長制限の 7 割近くに達している．また，このプロンプトを Gemini(Gemma) のトークナイザーで計測すると 106,875 トークンとなり，LLM によりトークン数に 2 割程度の差異があることが見て取れた．

プロンプトのサイズに関しては，プロンプト 1 から 4 のいずれにおいても，変種 full が最大となる．C はコールグラフが含まれないためにその分小さくなる．A は関数のソースコードが重複して含まれないために小さくなる．T はソースファイルが含まれないためにその分小さくなる．

(2) 対象プロダクトのソースファイルと比較してプロンプトのサイズがどの程度の大きさであるかを調べる．

表 9 に，プロンプト 1 から 4 について，含まれる Python ソースファイルの数，それらのソースファイルの行数の合計を示す．比較のために変種 full のプロンプトの行数も含めた．表では対象プロダクト全体の行数は約 22 万行であったため，対象プロダクトのソースコード全体を LLM に入力する場合と比較するとプロンプトの行数は 1/20 以下になっていることが分かる．

提案手法では対象プロダクトを実行して必要なソースファイルを特定するため，ユーザー（開発者）が手作業でソースファイルを選択する必要がない．仮に開発者がソースファイルを選択できたとしても，提案手法は関数単位でソースコードを抽出してプロンプトに付加するため，プロンプトをより小さくすることができる．例えば，プロンプト 1 の変種 full は行数で 22%(= 2549/11552) となっている．

5. まとめと展望

本研究では，ソースコードに関する問い合わせのための RAG 手法を提案した．提案手法では，ソフトウェアプロダクトの実行トレースからコールツリーと呼び出された関数のソースコードを抽出してプロンプトに付加する．これにより，プロダクトの機能の差異を調べたり，機能を実装するべき箇所を調べるといった，プロダクトの設計を考慮する必要がある問い合わせを可能にする．

実験では，再利用部分も含めると約 22 万行のオープンソースのプロダクトを対象として，ソフトウェア開発で想定される具体的な問い合わせについて，作成したプロンプトを LLM に入力してその応答を評価した．実験の結果，コールツリーとソースコードをプロンプトに含めることで応答の品質が向上する傾向がみられた．特に，関数が呼び出される順序をプロンプトに含めることが重要であることが分かった．一方で，LLM によっては，プロンプトのサイズが大きくなると応答の品質が低下する事例も確認された．

今後の課題としては，プロンプトの生成をより自動化するなどユーザーの手作業を軽減すること，ソフトウェアの多くのタスクに対応できるようプロンプトの作成方法を確立することが必要である．LLM のコンテキスト長制限に対処するための，より効果的な手法の検討も必要であろう．

謝辞　本研究は JSPS 科研費 22K11975 の助成を受けたものである．

文　献

[1] M. Jin, S. Shahriar, M. Tufano, X. Shi, S. Lu, N. Sundaresan, A. Svyatkovski, InferFix: End-to-End Program Repair with LLMs, ESEC/FSE 2023, pp. 1646–1656, 2023.
[2] M. Levy, A. Jacoby, Y. Goldberg, Same Task, More Tokens: the Impact of Input Length on the Reasoning Performance of Large Language Models, arXiv:2402.14848v1, 2024.
[3] P. Lewis, E. Perez, A. Piktus, et al., Retrieval-Augmented Generation for Knowledge-Intensive NLP Tasks, arXiv:2005.11401v4, 2021.
[4] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, B. Myers, Using an LLM to Help With Code Understanding, ICSE 2024, p. 881, 2024.
[5] M. Reid, N. Savinov, D. Teplyashin, et al., Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context, arXiv:2403.05530v1, 2024.
[6] C. S. Xia, Y. Deng, L. Zhang, Top Leaderboard Ranking = Top Coding Proficiency, Always? EvoEval: Evolving Coding Benchmarks via LLM, arXiv:2403.19114v1, 2024.
[7] J. Xu, Z. Cui, Y. Zhao, et al., UniLog: Automatic Logging via LLM and In-Context Learning, ICSE 2024, pp. 1–12, 2024.

（注4）：The Tokenizer Playground https://huggingface.co/spaces/Xenova/the-tokenizer-playground を利用した．