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

LLMを用いた協調的メタプログラミング手法に対するコード生成能力の再評価

—MetaGPTを対象として—

王　　天豪† 　柗本　真佑† 　楠本　真二†

† 大阪大学大学院情報科学研究科
E-mail: †{t-ou,shinsuke}@ist.osaka-u.ac.jp

あらまし　ChatGPT などの対話的 LLM を用いたメタプログラミングフレームワーク，MetaGPT が着目を浴びている．MetaGPT は複数の LLM エージェントの協調作業によって，与えられた自然言語の仕様からソースコードを自動的に生成する．MetaGPT の提案論文では，プログラミングコンテストのような簡単な問題に対しては約 8 割の成功率を達成したと報告している．MetaGPT 研究の課題の一つとして，性能評価が十分でないという点が挙げられる．第一に生成結果の妥当性を評価するためのテストが不十分であり，性能を過大評価している可能性がある．さらに同一問題に対するソースコードの生成試行回数が一度ずつのみであるため，LLM の持つランダムさを考慮できていない．本研究では MetaGPT の性能を再評価するために，妥当性評価のテストが追加されたコード生成データセットを用いた再実験を行う．また同一タスクに対して複数回の実験を行うことで，LLM のランダムさを考慮した性能評価を行う．

キーワード　LLM，メタプログラミング，コード生成，マルチエージェント，MetaGPT

1. はじめに

開発者がソースコードを直接記述せずにプログラミングを実現する，メタプログラミングというアイデアの実現性が増してきている．特に自然言語による対話が可能な大規模言語モデル（Large Language Model: LLM）を用いた，メタプログラミングフレームワークも登場している [1]．このフレームワークを用いることで，自然言語による抽象的なソフトウェアの要求を入力とするだけで，全自動で要求を満たすソースコードを得ることができる．

さらに，MetaGPT [1] や ChatDev [2] と呼ばれる複数の LLM モデルを用いたメタプログラミングフレームワークも存在する．これら 2 つの手法は人による開発を模倣しており，複数の LLM が特定の役割を持ったエージェントとして振る舞うことで，それらエージェント同士が協調作業することでソフトウェアを作り上げる．MetaGPT ではエージェントとして，プロダクトマネージャー，アーキテクト，プロジェクトマネージャー，エンジニア，QA エンジニアを設けている．複数エージェントによる協調作業の結果は，他の LLM ベース手法と比較して高い性能を持つことが報告されている．

しかしながら，MetaGPT 論文の評価方法に対しては 3 つの課題が存在している．まず，データセットに含まれる生成結果の成否判定用のテストが不十分であることが指摘されている [3]．テストの不十分さは，性能評価を過大評価につながっている可能性がある．また，評価方法として，複数の解を得るという状況を想定していない．LLM は一定のランダムさを内包しており，複数回の試行によって複数解を得ることが一般的である．この際には Pass@k と呼ばれる評価指標を用いることが一般的であるが，MetaGPT 論文では k を 1 に固定して実験を行っている．これは従来の正答率と等価な値であり，複数の解を考慮しているとは言えない．最後，MetaGPT の生成失敗の詳細分析が行われておらず，具体的にどのような失敗が発生していたかが一切明らかになっていない．

本研究の目的は，MetaGPT が持つコード生成能力の再評価である．そのために MetaGPT 論文が実施していた評価実験の方法を見直し，上記 3 つの課題を解決したうえでの再評価実験を行う．まず複数の解を取るという状況を想定するために，Pass@k の k を 1 から 10 まで変動させ，その際の性能を確かめる．テストが不十分という課題に対しては，テストが拡充された拡張データセットを採用することでその解決を試みる．これらの実験で得られた成否のうち，失敗ケースを目視により調査し，どのエージェントが失敗するのか，どういう原因で失敗するのかなどの詳細分析を行う．これにより具体的な失敗原因を明らかにする．

2. 準　　備

2.1 メタプログラミング

メタプログラミングとは，ソースコードの直接的な記述をせずにソースコードを獲得するプログラミング手法である [4]．メタプログラミングによって，ソフトウェア開発者は専門的な知識不要，かつ効率的にプログラムを作成できる．例えば，ローコードプラットフォーム [5] ではユーザの設計したフロー

チャートに基づき，対応するコードを自動的に生成する機能が存在している．このように，メタプログラミングは多様な場面での活用が進んでいる．

近年，大規模言語モデル（Large Language Model: LLM）を用いた，自然言語による対話ベースでのコード生成手法が着目を浴びている．GPT-4 [6] などのモデルは，大量の自然言語の文章にだけでなく大量のソースコードを学習に用いている．そのためユーザが与えた自然言語の要求に応じて，ソースコードの生成や書き換えといった様々なタスクに取り組むことも可能である．

2.2 複数の LLM を活用した手法

近年ではプロダクトコードの生成だけでなく，テストコードや仕様書等，あらゆるドキュメントを LLM に生成させる手法が検討されている．この手法では LLM に人間の役割（エージェント）を割り当て，複数のエージェントを相互に協調・連携させながらソフトウェア開発を行う．

例えば，Qian らは，ソフトウェア企業を模したフレームワーク ChatDev を提案している [2]．この手法では，複数の AI エージェントが最高経営責任者（CEO），最高技術責任者（CTO），プログラマ，テスターといった役職を担当する．これらのエージェントはあらかじめ設定されたプロンプト規則をもとに，相互にコミュニケーションを行い，ユーザ要求の詳細な自動分析やコード品質の自動的な評価を実現している．

2.3 MetaGPT

ChatDev や PairCoder とは異なる仕組みとして Hong ら [1] の MetaGPT がある．MetaGPT は複数の AI エージェントを用いた対話的かつ LLM ベースのメタプログラミングフレームワークである．

図 1 に MetaGPT の処理の流れを示す．MetaGPT ではウォーターフォール型の開発プロセスに基づき，プロダクトマネージャー，アーキテクト，プロジェクトマネージャー，エンジニア，QA エンジニア等の 5 種類のエージェントを割り当てている．エージェント間のコミュニケーションは様々なテキストドキュメントによって行われる． 図 1 ユーザが入力した自然言語の要求を入力として，プロダクトマネージャ（PM）が仕様書を作成し，その仕様書に基づいてアーキテクトがシステム設計書を作成する．最終的にエンジニアがソースコードを，QA エンジニアがテストを作成する．出力として得られるソースコードはテストによる検証が行われることになる．

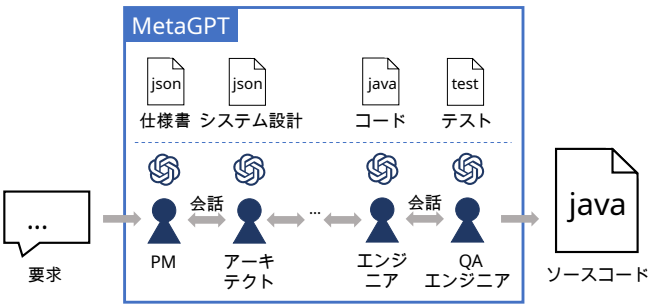

図 1　MetaGPT の外観図

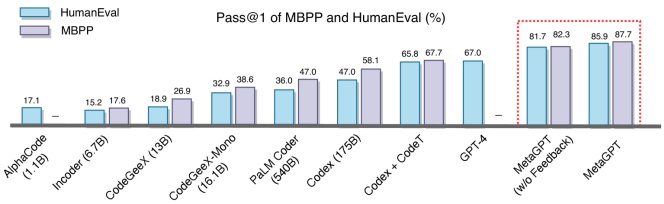

図 2　MetaGPT の評価実験の結果（文献 [1] より引用）

図 2 に MetaGPT の提案論文 [1] で示されていた，MetaGPT の性能評価実験の結果を示す．図は様々な LLM モデルに対するプログラム生成の性能を表しており，棒の高さが性能（Pass@k）である．2 つのコード生成のためのデータセット（MBPP [7] と HumanEval [8]）に対する実験結果である．図中の赤枠が MetaGPT の性能であり，フィードバックとは QA エンジニアの有無，すなわち生成したソースコードの動的な結果をエンジニアにフィードバックしているか否かを表している．比較対象手法としては，AlphaCode [9] などのコード生成特化型 LLM に加え，GPT-4 など汎用型 LLM も含まれている．MetaGPT は全ての他の手法と比べて高い精度を示している．またフィードバックを加えることで，MBPP に対しては正解率を 82.3 %から 87.7 %に，HumanEval に対しては 81.7 %から 85.9 %に向上している．

3. MetaGPT 研究の課題とその解決策

2.3 節でも述べた通り，MetaGPT は複数の AI エージェントによる全自動でのソフトウェア開発が可能であり，LLM 活用に対する様々な可能性を示している一方で，その評価方法について 3 つの課題が存在している．

- 生成結果の成否判定テストが不十分
- 異なる生成結果が生じる可能性を考慮していない
- 生成失敗の詳細分析が行われていない

本節では，各課題の詳細とその解決策について述べる．

3.1 生成結果の成否判定テストが不十分

MetaGPT 論文が評価実験に用いたデータセット MBPP と HumanEval に対しては，テストが不十分であると指摘されている [3]．Li ら [10] の研究によれば，MBPP と HumanEval には誤りやテストケースの不足が含まれており，一部のバグを適切に検出できない可能性が指摘されている．MBPP に含まれるテストケース数は 1 問題あたり平均 3 個，HumanEval は平均 7 個である．MBPP の問題が基本的なアルゴリズムに限定されている点を考慮に入れても，単体テストとしては不十分であり，MetaGPT のコード生成性能が過大に評価されるおそれがある．

そこで本研究では，生成されたコードを正確にテストするために，Liu らの提案する MBPP+ [3] と HumanEval+ [3] の 2 つのデータセットを利用する．これらのデータセットは，それぞれ MBPP および HumanEval に対して，より多くのテストケースを追加し，元のデータセットでは検出できなかったエラーを検出可能としている．本研究では MBPP+ と HumanEval+ を用いて，MetaGPT の生成結果の成否判定を行う．

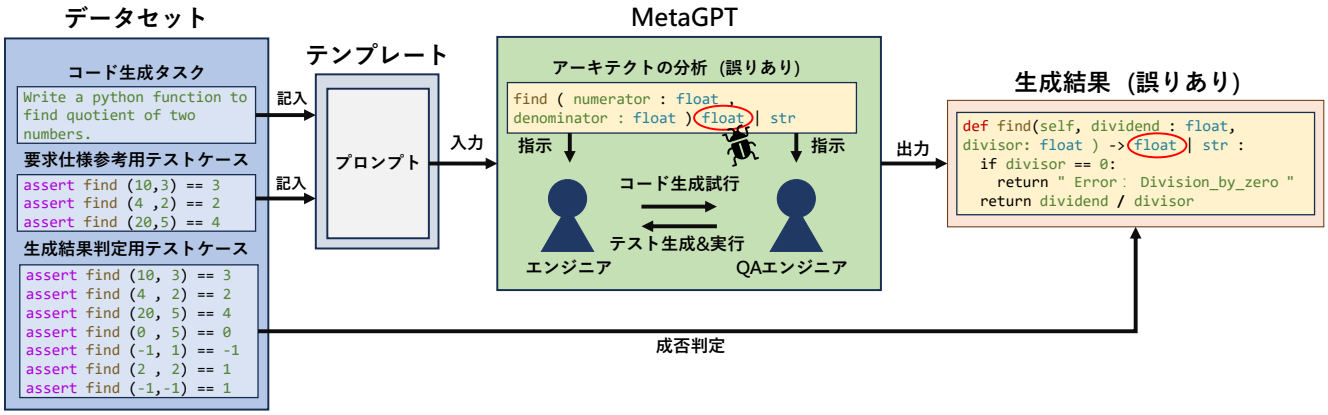

図 3　実験の流れの例（MBPP#292）

3.2　異なる生成結果が生じる可能性を考慮していない

MetaGPT 論文の評価実験では，同一の問題に対して複数の異なるコードを生成する可能性を考慮していない．LLM は Transformer モデル自体にランダム性があるため，同じ入力に対しても複数の解が得られる．よって LLM の性能評価においては，同一条件のタスクに対して複数の試行を行い，複数の解を得ることが一般的である．

MetaGPT 論文が採用していた評価指標は Pass$k$ である．この指標は LLM のための評価指標であり，LLM から得られた複数の解を k 個ピックアップした際に一つでも正解となる解を含む可能性を表している．解候補の中から k 個の無作為に抽出し，一つずつ目視確認して成否を確かめるというような状況を想定した指標である．

Pass$k$ 自体は LLM のランダムさ，すなわち複数回の試行を考慮に入れているが，MetaGPT 論文では k を 1 とした実験しか行っていない．k=1 とは，すなわち 1 個のピックアップで正解を引く可能性のことであり，単純に成功の割合（試行数に対して何個正解を含むか）と同値となってしまう．つまり複数の解を一切考慮に入れておらず，LLM を適用するという実践的な状況を想定していない．

この問題の解決のために，本研究では，様々な k に対して評価実験を行う．具体的には k の値を 1〜10 に変動させ，その際の Pass@k の値を確認する．基本的には k の値が大きくなるほど，正解を 1 個でも含む可能性（Pass@k 自体）は高くなるが，その成否の確認に要するコストは増大する．k を変動させることで，コストの増大に対する性能改善の程度を調べることができる．

3.3　生成失敗の詳細分析が行われていない

MetaGPT 論文の評価実験では，どのエージェントがエラーを誘発したのか，また MetaGPT が対応できない要求や条件があるのかといった詳細な分析が十分に行われていない．こうした情報が欠如していることで，MetaGPT のさらなる改善のための具体的な指針を得ることができない．

この問題の解決のために，本研究では MetaGPT の生成ログを目視により分析する．どのエージェントがどのような原因で最終的なコード生成の失敗を引き起こしたのかを分析することで，MetaGPT に存在する欠陥を明らかにすることができる．

4.　実　　験

4.1　実験の概要

前節で述べた MetaGPT 研究 [1] の評価の課題を見直し，MetaGPT の性能に対する再評価を行う．基本的には MetaGPT 研究の実験設定に従い，妥当でない，あるいは不十分な設定を改善する．実験の流れは次のとおりである．

- データセットから一つタスクを取り出す
- 当該タスクの要件とテストをプロンプトに埋め込む
- MetaGPT にプロンプトを渡しタスクに取り組むさせる
- MetaGPT の出力を得る
- プログラムの生成に成功したかを成否判定テストによって確認する

実験の流れの一例を 図 3 に示す．この例は MBPP というデータセットの 292 番目の問題であり，2 つの数値の商を計算する Python 関数を作成する，というプログラムの生成タスクである．MBPP はこのような基礎的なアルゴリズムを問う問題で構成されている．データセットには上記のタスクの内容に加えて，参考用のテストケースが含まれている．生成タスクと参考用のテストケースを，データセットで指定されたテンプレートにあわせてプロンプトとして組み立てる．その後，このプロンプトを MetaGPT に入力すると，各エージェントはそれぞれの役割に従って問題を分析し，開発文書やテスト，コードを生成する．この過程で誤りが発生すると，最終的に誤ったプログラムが生成される可能性がある．生成するプログラムが満たすべき要件をテストという形式で表現しているとも言える．なお，参考用テストとは独立に成否判定用のテストも存在している．これは MetaGPT に与えず，MetaGPT の出力の成否判定に用いる．よって成否の判定はテストを用いた全自動化が可能である．

表 1 に，本研究と MetaGPT 研究の実験設定の違いを示す．まず実験に用いるデータセットは，MetaGPT 研究ではプログラム生成用の評価データセットである MBPP と HumanEval であった．本研究ではこのデータセットのテストを拡充した MBPP+ と HumanEval+ を用いる．検証用テストは拡充前は 1

表 1　MetaGPT 研究と本研究の実験設定の比較

	先行研究 [1]	本研究
データセット	MBPP と HumanEval	MBPP+ と HumanEval+
#検証用テスト数	≈ 3 と ≈ 7	≈ 105 と ≈ 560
#タスク数	不明/974 と不明/164	20/974 と 20/164
評価指標	Pass@1	Pass@k
生成回数 n	不明	10
サンプル数 k	1	1〜10
失敗分析	N/A	目視により確認
モデル	GPT-4	GPT-4o

タスク当たり 3 個と 7 個であったのに対し，拡充後は 105 個と 560 個となっている．なお，本稿では各データセットに含まれる全タスクのうち，20 個ずつをランダムに選定し実験に用いる．MetaGPT 論文ではいくつのタスクが実験に用いられていたかは明確にされていない．

評価指標は Pass@1 のみを用いていたのに対し，本研究では k を 1〜10 に変動させ，その全ての評価値を確認する．この際の同一タスクに対する全試行回数は 10 回とする．タスク数と同様，MetaGPT 論文において試行回数を何回としていたかは記述されていない．

さらに失敗分析を目視により行う．特にどのエージェントがどのような理由で失敗しているのかを分類し，MetaGPT の性能改善に対する課題を洗い出す．この調査にあたっては MetaGPT が生成するログを目視で精査する．

なお評価モデルは GPT-4 ではなく GPT-4o を用いることとした．GPT-4o の性能は GPT-4 と同等かそれ以上であると指摘されている [11]．また，GPT-4o は GPT-4 より大幅に生成時間が短く，コストも安いモデルである．

4.2　データセット

本実験では，主に MBPP+ と HumanEval+ の 2 つのデータセットを使用した．MBPP+ および HumanEval+ は，MBPP および HumanEval を基に，自動テスト入力生成フレームワーク EvalPlus [3] を用いてそれぞれ 35 倍，80 倍のテストケースを追加したデータセットである．MBPP と HumanEval には，それぞれ 974 件と 164 件の簡単なコード生成タスクが含まれており，各タスクには問題文・参考解答・テストセットが付属している．より多くの成否判定テストを含むデータセットを使用し，同一の問題に対して複数回の生成を行うことで，MetaGPT のコード生成能力をより精確に評価することができる．

4.3　評 価 指 標

本実験では，MetaGPT が要求を満たすコードを生成する確率を評価するために Pass@k を用いる．Pass@k の計算方法は，eq. (1) に示すとおりである．ここで，n はコード生成を行った回数，c は要求を満たすコードが生成された回数，k はすべての生成結果のうちランダムに選択するサンプル数である．

$$\text{Pass@}k = \mathbb{E}_{\text{Problems}}\left[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}\right] \tag{1}$$

Pass@k の値はコード生成能力の高さを表しており，k を固定したとき，Pass@k の値が高いほど，正しくコードを生成できている可能性が高い．

自然言語処理の分野では，従来 ROUGE [12] や BLEU [13] といった指標が広く用いられてきた．これらの指標は LLM によって生成されたコードが正解コードと文字列レベルで一致しているかを評価している．しかし，コード生成の評価においては，コードが実現する機能の過不足やコードの正しさがより重要である．したがって，ROUGE や BLEU 等の指標より，適切なコード生成ができたかを表す Pass@k の方がコード生成指標として適している [14]．

表 2　生成結果の成否判定テスト通過率

データセット	生成回数	成否判定テスト通過率
MBPP	200	67.0 %
MBPP+	200	61.0 %
HumanEval	200	61.0 %
HumanEval+	200	52.5 %
MBPP と HumanEval	400	64.0 %
MBPP+ と HumanEval+	400	56.5 %

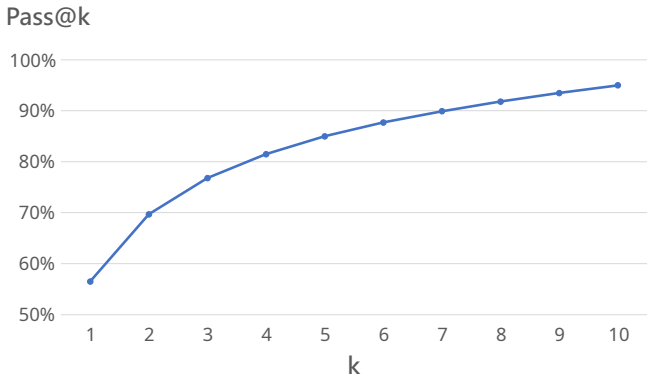

図 4　MBPP+ と HumanEval+ における生成コードの平均 Pass@k

5.　結　　果

5.1　成否判定テスト通過率による生成能力の評価

表 2 は，成否判定テストのテストケースの追加前後で，MetaGPT が生成したコードがコード生成タスクの成否判定テストを通過できる割合を示している．4.2 節で述べたとおり，MBPP+ および HumanEval+ の「+」は，MBPP および HumanEval と同じコード生成タスクを含みつつ，それらに大量のテストケースを追加したものである．テストケースを追加することにより，MetaGPT が生成したコードがタスクの成否判定テストを通過する割合は，MBPP での 67 %から MBPP+ での 61 %に低下した．また，MetaGPT が生成したコードがタスクの成否判定テストを通過する割合は，HumanEval での 61 %から HumanEval+ での 52.5 %に低下した．

MetaGPT 論文の評価実験 [1] では，MetaGPT の平均正解率が 2 つのデータセットで約 86 %と報告されている．しかし，本実験の結果から，MetaGPT が過大に評価されている可能性があることが示唆された．

表 3　エージェントごとの失敗原因の分類

エージェント	テスト失敗原因の種類	個数	割合
プロダクトマネージャー	関数の入力の解釈に失敗	14	8.1 %
プロダクトマネージャー	小計	14	8.1 %
アーキテクト	関数の出力の型が正しくない	24	13.9 %
アーキテクト	アルゴリズム設計が正しくない	9	5.2 %
アーキテクト	小計	33	19.1 %
エンジニア	設計通りにコードを生成できない	57	33.0 %
エンジニア	特殊な状況の対応できない	42	24.3 %
エンジニア	返り値の文字列が僅かに違う	15	8.7 %
エンジニア	ループの範囲が違う	7	4.0 %
エンジニア	返り値が設計通りにソートされていない	5	2.9 %
エンジニア	小計	126	72.8 %
エージェント全体	合計	174	100.0 %

5.2　Pass@k による生成能力の評価

図 4 は，2 つのデータセットにおける k が 1 から 10 の値を取る場合の平均 Pass@k の値を示している．k=1，つまり 1 回の生成で正しいコードが得られる確率を考える場合，MetaGPT の平均 Pass@1 は 56.5 %であり，MetaGPT 提案論文における 86 %と比較して大幅に低い．また，MBPP+ と HumanEval+ に含むコード生成タスクは簡単なプログラミング問題であるにもかかわらず，1 回の試行で正しいコードを生成する確率は 56.5 %であり，MetaGPT には改善の余地があると考えている．

k=3 の場合，生成されたコードに正解が含まれる可能性は約 76.8 %に達する．これは完全自動で生成されたコードであることを考慮すると，精度が良いと言える．

5.3　失敗原因の分析

表 3 では，173 回の失敗において，どのエージェントがどの原因による失敗を引き起こしたことを示している．MetaGPT はウォーターフォールモデルを模倣してコード生成を行っているため，上流設計の誤りが下流の開発工程にも波及する．そのため，本稿では最初にエラーを引き起こしたエージェントのみを集計している．173 回の失敗のうち，14 回（8.1 %）はプロダクトマネージャー，33 回（19.1 %）はアーキテクト，126 回（72.8 %）がエンジニアによるものであった．以下では，一部の失敗原因を例に挙げ，それらがどのように発生したかを詳しく説明する．

図 5 は，プロダクトマネージャーが関数の入力を誤って分析し，最終的に誤ったコードが生成された例を示している．生成タスクでは，円錐の側面積を計算することを要求しており，3 つのテスト用例が参考として提示されている．本来，関数の入力は円錐の半径と高さであるべきだが，プロダクトマネージャーは誤って，ユーザーが半径と斜高にもとづいて側面積を計算することを望んでいると判断してしまった．前述のとおり，上流設計を担当するエージェントの誤りは，後続のエージェントが参照する設計文書にも誤りを連鎖的に広める．そのため，プロダクトマネージャーの開発文書を基にアルゴリズム設計やコード生成を行った他のエージェントが最終的に生成した関数は，半径と斜高を入力とするようになってしまい，成否判定テストを通過できなかった．

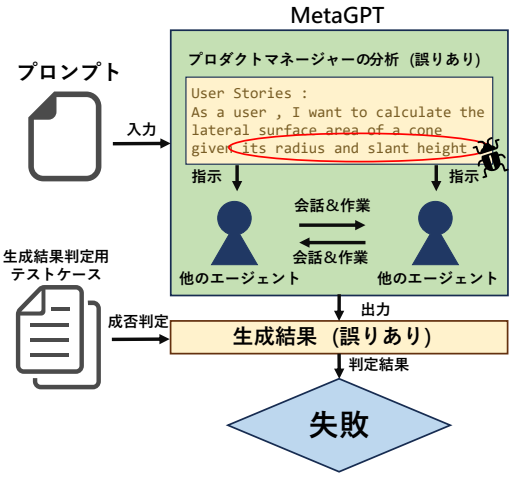

図 5　関数入力の誤解による失敗の例

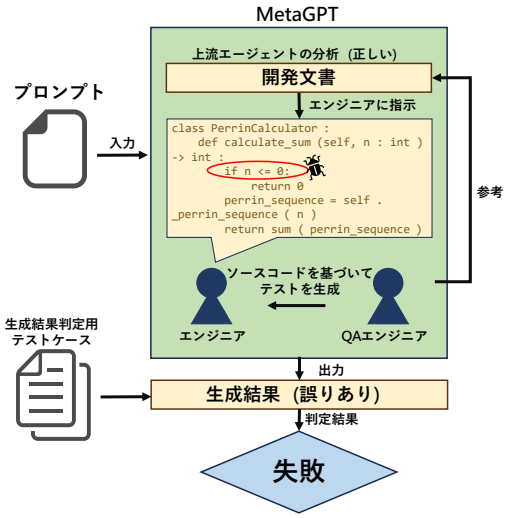

図 6　エンジニアによる失敗の例

図 6 は，上流設計が正しく行われたにもかかわらず，エンジニアがループの範囲を誤った結果，誤りが生じた例を示している．生成タスクでは，第 n 番目のペリン数を求めるよう要求され，参考テスト用例では第 9，第 10，第 11 のペリン数が与えられている．上流設計を担当するエージェントが問題

表 4　誤ったコードを生成した原因

失敗原因	個数	割合
LLM のコード生成能力不足	122	70.5 %
プロンプト中の情報の見落とし	51	29.5 %

を正しく分析していたにもかかわらず，エンジニアはペリン数を第 0 番目ではなく，第 1 番目から計算するものと誤解してコードを生成してしまった．さらに，QA エンジニアもエンジニアのコードと考え方をもとにテスト用例を生成したため，このバグを検出できるテストを作成しなかった．結果として，正しいペリン数を返すはずの関数が誤った値を返すコードとなり，テストを通過できなかった．

また，特殊な状況の対応できないことの一例を示す．単語中の母音の数を計算する際に，語末の y を母音としてカウントするよう明記されているにもかかわらず，MetaGPT はこれを無視したコードを生成した．MetaGPT 内部のプロンプトシステムをさらに最適化し，エージェント間の通信方式を改善することで，この種の問題を低減できると考えられる．

表 4 に示すように，LLM のコード生成能力不足は誤ったコードを生成した主要な原因である．MetaGPT が内部で利用する LLM を変更することで，この問題がある程度緩和される可能性がある．プロンプト中の情報の見落としについては，MetaGPT 内部のエージェント定義プロンプトやエージェント間の会話方法を最適化することで，この問題がある程度緩和される可能性がある．

6. 妥当性への脅威

MetaGPT の提案論文で使用された MetaGPT は，バージョン 0.4 を基に，一部のプロンプトを修正して得られた特別なバージョンである．しかし，このバージョンは著者によって公開されておらず，さらにユーザは常に最新バージョンを利用する傾向があるため，本研究では MetaGPT のバージョン 0.8.1 を使用している．異なるバージョンの MetaGPT を使用した場合，実験結果が異なる可能性がある．

また，本実験で MetaGPT が使用する LLM は GPT-4o である．LLM のランダム性や，OpenAI による GPT-4o のアップデートが生成結果に影響を与える可能性があり，同一の生成タスクでも異なるコードが生成される場合がある．さらに，異なる LLM モデルを使用した場合にも生成結果が変わる可能性がある．

さらに，本実験では金銭および時間の制約により，MBPP+ および HumanEval+ からそれぞれランダムに 20 個の生成タスクを選択し，すべての問題をテストしているわけではない．異なる生成タスクをランダムに選択した場合や，データセット全体を使用した場合には，異なる実験結果が得られる可能性がある．

7. おわりに

本研究では，MetaGPT のコード生成能力の再評価を行った．評価の結果として，MetaGPT の正解率は 86 %ではなく 56 %である．コード生成性能を過大評価されている．Pass@*3* は 76.8 %，つまり解を 3 つ選べば 76.8 %正解を含む．また，生成失敗の原因分析について，失敗原因の 72.9 %はエンジニアのミス．タスクの理解というよりコード生成自体の難しさがある．

今後の課題として，失敗原因のさらなる分析が挙げられる．アーキテクトが誤ったアルゴリズムを設計する際の傾向やエンジニアがコード構造設計を失敗しやすい状況を分析が必要である．また，今回の調査ではデータセットの合計 1,138 問のうち 40 問のみ実験しているため，データセットに含まれる全テストに対する実験も挙げられる．

謝辞　本研究の一部は，JSPS 科研費（JP24H00692, JP21H04877, JP21K18302）による助成を受けた．

文　献

[1] S. Hong, X. Zheng, J. Chen, Y. Cheng, J. Wang, C. Zhang, Z. Wang, S.K.S. Yau, Z. Lin, L. Zhou, et al., “Metagpt: Meta programming for multi-agent collaborative framework,” https://arxiv.org/abs/2308.00352, 2023.

[2] C. Qian, W. Liu, H. Liu, N. Chen, Y. Dang, J. Li, C. Yang, W. Chen, Y. Su, X. Cong, et al., “Chatdev: Communicative agents for software development,” Association for Computational Linguistics, pp.15174–15186, 2024.

[3] J. Liu, C.S. Xia, Y. Wang, and L. Zhang, “Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation,” Advances in Neural Information Processing Systems, vol.36, pp.21558–21572, 2024.

[4] J.R. Cordy and M. Shukla, Practical metaprogramming, Queen’s University of Kingston. Department of Computing and Information Science, 1992.

[5] Y. Luo, P. Liang, C. Wang, M. Shahin, and J. Zhan, “Characteristics and challenges of low-code development: the practitioners’ perspective,” Proceedings of international symposium on empirical software engineering and measurement, pp.1–11, 2021.

[6] J. Achiam, S. Adler, S. Agarwal, L. Ahmad, I. Akkaya, F.L. Aleman, D. Almeida, J. Altenschmidt, S. Altman, S. Anadkat, et al., “Gpt-4 technical report,” https://arxiv.org/abs/2303.08774, 2023.

[7] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al., “Program synthesis with large language models,” https://arxiv.org/abs/2108.07732, 2021.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H.P.D.O. Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, et al., “Evaluating large language models trained on code,” https://arxiv.org/abs/2107.03374, 2021.

[9] D. Huang, Q. Bu, J.M. Zhang, M. Luck, and H. Cui, “Agentcoder: Multi-agent-based code generation with iterative testing and optimisation,” https://arxiv.org/abs/2312.13010, 2023.

[10] Y. Li, D. Choi, J. Chung, N. Kushman, J. Schrittwieser, R. Leblond, T. Eccles, J. Keeling, F. Gimeno, A. Dal Lago, et al., “Competition-level code generation with alphacode,” Journal on Science, vol.378, no.6624, pp.1092–1097, 2022.

[11] R. Islam and O.M. Moushi, “Gpt-4o: The cutting-edge advancement in multimodal llm,” http://dx.doi.org/10.36227/techrxiv.171986596.65533294/v1, 2024.

[12] C.-Y. Lin, “Rouge: A package for automatic evaluation of summaries,” Text summarization branches out, pp.74–81, 2004.

[13] K. Papineni, S. Roukos, T. Ward, and W.-J. Zhu, “Bleu: a method for automatic evaluation of machine translation,” Association for Computational Linguistics, pp.311–318, 2002.

[14] S. Kulal, P. Pasupat, K. Chandra, M. Lee, O. Padon, A. Aiken, and P.S. Liang, “Spoc: Search-based pseudocode to code,” Advances in Neural Information Processing Systems, vol.32, pp.11906–11917, 2019.