修士学位論文

題目

ミューテーション解析に基づく量子プログラム自動修復手法の修正成功率と説明性の向上

指導教員

肥後 芳樹 教授

報告者

吉田 千優

令和 8 年 2 月 2 日

大阪大学 大学院情報科学研究科

コンピュータサイエンス専攻 ソフトウェア工学講座

令和 7 年度 修士学位論文

ミューテーション解析に基づく量子プログラム自動修復手法の修正成功率と説明性の向上

吉田 千優

内容梗概

近年，量子プログラムを対象とした自動プログラム修復 (Automated Program Repair: APR) 手法が提案されている．しかし，既存手法は修正成功率が低い，あるいは生成される修正内容の理解が難しいといった課題を抱えている．本研究では，大規模言語モデル (Large Language Model: LLM) がプログラムの修正コードとともに，適用した修正内容の自然言語による説明を生成する枠組みを構築する．量子プログラムに対する APR において，プロンプトに含める文脈情報が性能に与える影響を調査するため，静的情報，動的情報，およびミューテーション解析結果の組み合わせが異なる 8 種類のプロンプト構成を設計した．静的情報は，バグを含むコードやバグの説明といった，プログラムを実行することなく得られる情報である．動的情報は，プログラム実行時のエラーメッセージまたはテスト結果のいずれかである．ミューテーション解析は，プログラムの一部に小さな変更を加えた際の実行結果の変化を評価する手法であり，スタックトレースなどの単純な実行結果と比べて，より詳細な動的情報を提供する．実験の結果，動的情報とミューテーション解析結果を組み合わせることで修正成功率が向上し，本実験では 94.4% を達成した．さらに，ミューテーション解析結果を用いることで，バグの位置情報に関する記述がより正確かつ網羅的で簡潔になり，生成される説明の品質が向上することを確認した．これらの結果から，ミューテーション解析は，量子プログラムに対する LLM ベース APR において有用な文脈情報を提供し，修正成功率と説明品質の双方の向上に寄与することが示された．

主な用語

量子プログラム
自動プログラム修復
ミューテーション解析

大規模言語モデル

説明性

目次

1 はじめに 5

2 背景 8

2.1 量子コンピューティング 8

2.2 ミューテーション解析 8

3 関連研究 9

3.1 古典プログラムにおける自動プログラム修復 9

3.2 量子プログラムにおける自動プログラム修復 10

3.3 パッチの説明生成 10

4 実験 12

4.1 目的 12

4.2 バグベンチマーク 14

4.3 ミューテーション 15

4.4 プロンプト構成 17

4.5 大規模言語モデル 19

4.6 評価指標 20

4.7 実験環境 21

5 実験結果 23

5.1 RQ1: ミューテーション解析結果は量子プログラム APR の修正成功率にどのような影響を与えるのか 23

5.2 RQ2: ミューテーション演算子の違いは量子プログラム APR の修正成功率にどのような影響を与えるのか 24

5.3 RQ3: ミューテーション解析結果は量子プログラム APR の説明性にどのような影響を与えるのか 25

6 考察 27

6.1 プロンプト構成間における修復成功プログラムの差異 27

6.2 ミューテーション解析と修復結果の関係に関する事例分析 27

7 妥当性への脅威 32

7.1 構成妥当性 32

7.2 内的妥当性 32

7.3 外的妥当性 32

8 おわりに 33

謝辞 34

参考文献 36

付録 41

1　はじめに

量子コンピューティングは，量子力学の原理を利用することで，特定の計算課題において古典計算機を上回る性能を示す可能性が指摘されている [1]．量子コンピュータの計算能力を活用するためには，量子ビットに対する一連の操作を記述した量子プログラムを記述する必要がある．

古典 (非量子) プログラムと同様に，量子プログラムにおいてもデバッグが必要である [2, 3]．しかし，量子プログラムのデバッグは，量子特有の概念を扱う必要があるため，固有の困難さを伴う [4, 5, 6]．量子プログラムのデバッグを支援するため，研究者らは量子ソフトウェアに特化した自動プログラム修復 (Automated Program Repair: APR) 手法を提案してきた [7, 8, 9]．APR の目的は，バグを含むプログラムに修正を加え，与えられたテストスイート内のすべてのテストケースを通過させる修正パッチを自動生成することである [10]．Guo らは，ChatGPT [11] の量子プログラムを修復する能力について調査した [8]．その結果，量子アルゴリズムの実装を含む複雑なバグに対しては，ChatGPT が修復できた割合は 17% にとどまることが示された．Tan らおよび Li らは，量子ゲートを修正パッチとして生成し，量子プログラムに挿入する APR 手法を提案した [9, 7]．現在の最先端手法である HornBro は，ChatGPT ベースの APR 手法と比較して，より多くのバグを修復できることが示されている [9]．しかし，この性能向上は，追加の量子ゲートを挿入することにより，プログラム複雑度が増大するという代償を伴う．実際，ある事例では，HornBro がバグ修復の過程で最大 249 個の追加ゲートを挿入し，修正後のプログラムの可読性および保守性を低下させることが報告されている [12]．このように，量子プログラムを対象とした既存の APR 手法は，(1) 修正成功率が低い，または (2) 修正後のプログラムの可読性・保守性が低い，といういずれかの課題を抱えている．

本研究では，ミューテーション解析から得られる知見を大規模言語モデル (Large Language Model: LLM) による修復プロセスに組み込むことで，量子プログラムに対する APR の性能向上を目指す．ミューテーション解析は，プログラムの特定箇所に小さな変更を加えた際に，その実行結果がどのように変化するかを評価する手法である．量子プログラムに対するミューテーション解析では，算術演算子の置換といった古典プログラムにも適用可能なミューテーション演算子に加え，量子ゲートの追加や削除といった量子特有のミューテーション演算子を

組み合わせて用いる [13]．このアプローチは，テスト [13, 14, 15, 16] や欠陥限局 [17] の分野において有効性が示されている．本研究の実験では，まず各プログラムに対してミューテーション解析を適用する．その後，バグを含む量子プログラム，そのスタックトレース，およびミューテーション解析結果を LLM への入力として与え，修正後のコードとともに，適用した修正内容の自然言語による説明を生成させる．修正コードと併せて自然言語による説明を生成することで，修正後のコードだけでは把握しにくい修正の意図や根拠を，開発者が理解しやすくなる．本研究では，ミューテーション解析結果が，スタックトレースのような単純な実行結果よりも詳細な動的情報を提供することにより，修正成功率の向上に寄与するとともに，より充実した説明の生成を可能にするという仮説を立てる．また，適切なミューテーション演算子を選択することが，量子プログラムの修正成功率に影響を与えるのではないかと考えた．本研究の研究課題は以下の通りである．

RQ1 **(修正成功率)** ミューテーション解析結果は量子プログラム APR の修正成功率にどのような影響を与えるのか

RQ2 **(修正成功率)** ミューテーション演算子の違いは量子プログラム APR の修正成功率にどのような影響を与えるのか

RQ3 **(説明性)** ミューテーション解析結果は量子プログラム APR の説明性にどのような影響を与えるのか

本研究の実験では，Bugs4Q データセット [18] に含まれる 18 件の実世界のバグを含む量子プログラムを対象とした．これらのプログラムはすべて，量子プログラミングで広く利用されているライブラリである Qiskit [19] を用いて Python で記述されている．実験全体を通して，プログラム修復の実行においては最先端の LLM である GPT-5 [20] を基盤モデルとして用いた．ミューテーション解析には，量子プログラム向けに設計されたミューテーションテストフレームワークである QMutPy [13] を使用した．異なる文脈情報が修復性能に与える影響を分析するため，8 種類のプロンプト構成を比較した．本研究の主な貢献は以下のとおりである．

- ミューテーション解析が，量子プログラムに対する LLM ベース APR において有効な情報源となり得ることを示す，初の実証的知見を提示した．
- 動的実行情報とミューテーション解析結果の双方をプロンプトに組み込むことで，修正成功率が向上することを明らかにした．

- 量子ミューテーション演算子，もしくは古典ミューテーション演算子のいずれかではなく，これら両方を利用したミューテーション解析結果を用いることが，修正成功率を高めることを明らかにした．
- ミューテーション解析結果を活用することで，バグの位置に関する説明がより正確かつ網羅的で簡潔になり，LLM が生成する説明の品質が向上することを示した．

2　背景

2.1　量子コンピューティング

量子コンピューティングは，特定の計算課題において古典 (非量子) の計算機を上回る性能を発揮する可能性があることから，大きな注目を集めている [1]．想定される応用分野には，暗号 [21]，化学シミュレーション [22]，機械学習 [23] などが含まれる．量子コンピュータの計算能力を活用するためには，開発者は量子ビット (*qubit*) に対する一連の操作を記述した量子プログラムを作成する必要がある．ここで，各操作は量子ゲートと呼ばれる．

古典プログラムと同様に，量子プログラムにおいてもその正しさを保証するためにデバッグが必要である [2, 3]．しかし，量子プログラムのデバッグは，重ね合わせやエンタングルメントといった量子特有の概念を扱う必要があるため，固有の困難さを伴う [4, 5, 6]．実際，これらの概念は，量子特有の技術的負債 [24, 25] やコードスメル [12] といった，従来とは異なるソフトウェア工学上の課題を引き起こすことが指摘されている．

2.2　ミューテーション解析

プログラムに対して，算術演算子の置換といった小規模な構文的変更を加える操作をミューテーションと呼び，その変更が適用されたプログラムをミュータントと呼ぶ．ミューテーション解析は，多数のミュータントを体系的に生成し，それらに対するテスト実行結果の変化を分析することで，テストスイートの有効性やプログラムの振る舞いを評価する手法である [26]．量子プログラムに対するミューテーション解析では，算術演算子の置換といった古典プログラムにも適用可能なミューテーション演算子に加え，量子ゲートの追加や削除といった量子特有のミューテーション演算子を組み合わせて用いる [13]．このアプローチは，テスト [13, 14, 15, 16] や欠陥限局 [17] の分野において有効性が示されている．

また，多様なミューテーション演算子の中から，プログラムの性質に応じて適切な演算子を特定・選択することは，解析の効率化と精度の向上の観点から極めて重要である．Mendiluze らは，大規模な実証調査を通じて，ミューテーション演算子の違いがミュータントの検出難易度に大きな影響を与えることを示し，評価目的に応じた演算子選択の重要性を指摘している [27]．

3　関連研究

本節では，古典プログラムおよび量子プログラムの双方を対象とした自動プログラム修復 (APR) に関する既存研究の全体像を示す．また，修正パッチの自然言語による説明生成の関連研究についても議論する．

3.1　古典プログラムにおける自動プログラム修復

APR 手法は，大きく以下の 3 種類に分類される [10].

1. **ヒューリスティックベース修復**：修正パッチを反復的に生成・検証する手法
2. **制約ベース修復**：修復問題を制約充足問題として定式化する手法
3. **学習ベース修復**：修正パッチを生成する予測モデルを構築する手法

これらの中でも，近年では機械学習，特に大規模言語モデル (LLM) の急速な発展を背景として，学習ベース修復 [28] が注目を集めている．Xia らは，LLM を APR に直接適用し，Java, Python，および C を対象として，LLM ベースの修復が既存の APR 手法を上回る性能を示すことを明らかにした [29]．LLM ベースのプログラム修復に関する多くの研究は，プロンプトに含める文脈情報の重要性を指摘している [30, 31, 32, 33]．例えば，InferFix は，静的解析の結果やバグ位置の情報をプロンプトに付加することで，LLM がバグの種類やコード中での位置を明示的に認識したうえで修復を生成できるようにしている [30]．Ehsani らは，バグレベル，リポジトリレベル，およびプロジェクトレベルの知識を階層的にプロンプトへ注入することで，より広範かつ関連性の高い修復文脈を提供する手法を提案している [33]．Bouzenia らは，動的なプロンプト生成を採用したエージェント指向システムである RepairAgent を提案しており，必要に応じてエージェントが LLM プロンプトに含める情報を判断し，ツール呼び出しを通じてプロンプトを動的に更新する [32].

本研究は，LLM ベースの自動プログラム修復 (APR) において，プロンプトが性能に与える影響を調査するという点で，先行研究と同様の方向性に位置づけられる．一方で，本研究の新規性は，量子プログラムに対するテスト [13, 14, 15, 16] および欠陥限局 [17] において有効性が示されているミューテーション解析結果を，LLM へのプロンプトに組み込む点にある．量子プログラムおよび古典プログラムのいずれにおいても，LLM ベース APR の性能向上を目

的としてミューテーション解析を活用した先行研究は確認されていない．

3.2 量子プログラムにおける自動プログラム修復

量子プログラムのバグ修正は，量子コンピューティングに関する専門的知識を必要とするため難易度が高い [4, 5, 6]．Guo らは，ChatGPT [11] を用いて量子プログラムに対する APR の可能性を検討した [8]．その結果，対象データセットに含まれる古典 (非量子) バグについては 97% を修復できた一方で，量子バグについては 17% しか修復できなかったことが報告されている．Li らは，ユニタリ演算を自動生成することで量子プログラムを修復する手法である UnitAR を提案した [7]．Tan らは，バグの原因となる誤った量子ゲートを除去し，新たなゲートを合成することで正しい振る舞いを実現する手法である HornBro を提案した [9]．本研究で用いたベンチマークにおいても，HornBro は ChatGPT ベースの APR より多くのバグを修復できることが示されている．UnitAR および HornBro はいずれも，新たな量子ゲートを生成・挿入することでバグを修復する点から，合成ベース修復手法と位置づけられる．本研究では，以下の理由から LLM ベースの修復手法を採用する．

1. 合成ベース修復手法は主に量子ゲートに起因するバグに限定されるのに対し，LLM ベース手法はより柔軟であり，実世界の量子プログラムにおいても頻出する API 関連バグなど，より広範な種類のバグに対応可能である [5].
2. 合成ベース修復手法は量子プログラム中のゲート数を増加させ，結果としてプログラムの複雑度を高める．過剰なゲート数は量子コードスメルの一種としても指摘されており [12]，これを避けることは Google Quantum AI においてベストプラクティスとして推奨されている [34].

本研究の手法では，LLM に対して最小限の修正を生成させるだけでなく，それらに対する自然言語による説明の生成も指示する．このように修正内容の説明を併せて生成することで，他の量子プログラム向け APR 手法とは異なり，開発者が各変更の意図や根拠を確認・検証できるようになる．

3.3 パッチの説明生成

パッチの説明は，修正内容の理解やレビューを支援する重要な役割を担う．Liang らは，オープンソースプロジェクトにおけるコミットメッセージやレビューコメントを分析し，開発者が

記述するパッチ説明には単なるコード変更の要約だけでなく，不具合の原因や修正の意図といった背景情報が含まれることを明らかにした [35].

近年は，大規模言語モデル (LLM) を用いて，パッチと同時に自然言語による説明を自動生成する研究が行われている．Sobania らは，LLM が生成したソフトウェアパッチの説明を対象に，人間による評価を通じてその品質を分析した [36]．その結果，生成された説明は一定の有用性を持つ一方で，誤った原因説明や不完全な記述が含まれる場合があることを報告している．また，説明品質はモデルに与える文脈情報に大きく依存する可能性が示唆されている．

説明の有用性を議論するためには，その評価方法も重要である．Nauta らは，説明可能な AI(Explainable AI: XAI) における説明評価手法を体系的に整理し，従来の研究が事例ベースの定性的評価に偏っている点を指摘した [37]．同研究では，説明の正確性，完全性，複雑性といった定量的指標を用いた評価の必要性が強調されている．本研究では，先行研究で議論されてきたパッチ説明の評価観点に基づき，量子プログラムに対する LLM ベース APR で生成される修正パッチの説明を評価する．これにより，説明の評価における主観的判断の影響を可能な限り抑えた説明性評価を目指す．

4 実験

本節では，実験の目的，ベンチマーク，プロンプト構成，および使用する LLM を含む，本研究の実験設計について説明する．本研究の目的は，ミューテーション解析結果を LLM ベースの修復に組み込むことが，修正成功率および生成される説明の品質にどのような影響を与えるかを明らかにすることである．図 1 に実験プロセスの概要を示す．バグを含む各プログラムに対して 8 種類の異なるプロンプト構成を用意する．いずれのプロンプト構成においても，LLM には修正後のコードと修正内容に関する説明の双方を生成するよう指示する．

4.1 目的

本研究では，ミューテーション解析結果を大規模言語モデル (Large Language Model: LLM) による量子プログラム修復プロセスに組み込むことで，自動プログラム修復 (APR) の性能および生成される説明の品質にどのような影響を与えるかを調査する．特に，ミューテーション解析結果が，スタックトレースのような単純な実行結果よりも詳細な動的情報を提供することで，修正成功率の向上に寄与するとともに，より質の高い説明生成を可能にするという仮説を検証する．また，ミューテーション解析の実行時に利用する演算子の違いが，量子プログラムの修正成功率に影響を与えるかどうかも調査する．

この目的のため，本研究では以下の 3 つの研究課題 (Research Questions: RQs) を設定する．

RQ1 **(修正成功率)** ミューテーション解析結果は量子プログラム APR の修正成功率にどのような影響を与えるのか

RQ2 **(修正成功率)** ミューテーション演算子の違いは量子プログラム APR の修正成功率にどのような影響を与えるのか

RQ3 **(説明性)** ミューテーション解析結果は量子プログラム APR の説明性にどのような影響を与えるのか

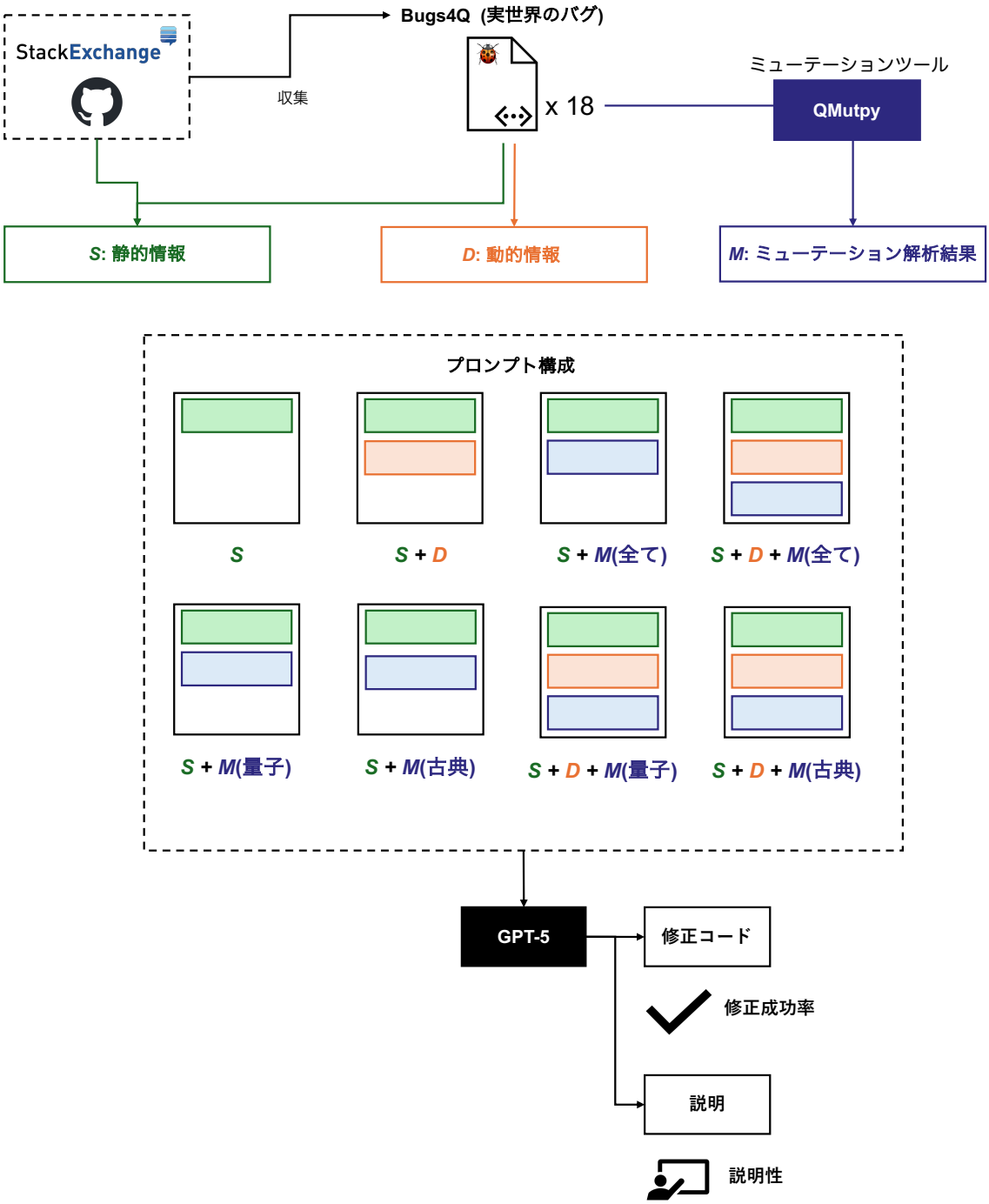
StackExchange
収集
Bugs4Q (実世界のバグ)
x 18
ミューテーションツール
QMutpy
S: 静的情報
D: 動的情報
M: ミューテーション解析結果
プロンプト構成
S
S + D
S + M(全て)
S + D + M(全て)
S + M(量子)
S + M(古典)
S + D + M(量子)
S + D + M(古典)
GPT-5
修正コード
修正成功率
説明
説明性

図 1: 実験の概要

4.2 バグベンチマーク

本研究では，Qiskit [19] を用いて実装された量子プログラムにおける実世界のバグを集めたベンチマークである Bugs4Q [18] を使用する．

Bugs4Q には，GitHub，Stack Overflow，および Stack Exchange から収集された 42 件のバグを含む量子プログラムと，それらに対応する修正済みプログラムが含まれている．本研究で Bugs4Q を採用した理由は，以下の 2 条件を満たしているためである．

1. 人間の開発者によって記述された実世界の量子プログラムに含まれるバグが収集されていること．
2. バグを含むコードおよび修正済みコードの動作を検証するためのテストコードが含まれていること．

第 1 の条件は，LLM が生成した修正に対する説明と，開発者が実際に意図した修正内容とを比較するために必要である．第 2 の条件は，ミューテーション解析を実施するとともに，生成された修正パッチが正しいかどうかを検証するために不可欠である．

Bugs4Q に含まれる 42 件のプログラムのうち，本研究では 18 件を実験対象として選定した．まず，Bugs4Q のレプリケーションパッケージ[*1]をクローンし，各バグが再現可能であるかを確認した．この確認の結果，バグを再現できなかった 19 件のプログラムを除外した．Bugs4Q に含まれるバグは，Qiskit ライブラリによって例外が送出される場合のような例外発生型 (*Throw Exception: TE*) と，測定された量子状態が期待される状態と異なる場合に AssertionError が発生する場合のような誤った出力型 (*Wrong Output: WO*) の 2 種類の症状に分類される．TE 型のバグは 8 件，WO 型のバグは 10 件であった．さらに，実験で用いるプロンプトを構成するために必要な以下の条件のいずれかを満たさない 5 件のプログラムを除外した．

1. ミューテーション解析を実行した際に，少なくとも 1 つのミュータントが生成されること．
2. 元となるソースコードのリポジトリ URL にアクセス可能であること．

[*1] `https://github.com/Z-928/Bugs4Q-Framework`

4.3 ミューテーション

本研究では，バグを含む各プログラムに対して QMutPy [13] を適用し，ミューテーション解析結果を取得した．QMutPy は，量子ゲートの挿入，削除，および置換に加え，量子測定の挿入および削除を含む，20 種類の古典ミューテーション演算子と 5 種類の量子ミューテーション演算子を実装している．このうち，量子ゲートの挿入および量子測定の挿入は，本研究の解析対象から除外した．1 回のミューテーションで，各ミューテーション演算子がプログラムの 1 箇所に対して適用される．ミューテーション演算子は反復的に適用され，各量子プログラムに対して複数の変異プログラム (ミュータント) が生成される．ミューテーション演算子の一覧を，表 1 に示す．

表 1: ミューテーション解析に利用したミューテーション演算子一覧

演算子	内容
AOD	算術演算子の削除 (Arithmetic Operator Deletion)
AOR	算術演算子の置換（Arithmetic Operator Replacement)
ASR	代入演算子の置換 (Assignment Operator Replacement)
BCR	break 文と continue 文の置換 (Break Continue Replacement)
COD	条件演算子の削除 (Conditional Operator Deletion)
COI	条件演算子の挿入 (Conditional Operator Insertion)
CRP	定数の置換 (Constant Replacement)
DDL	デコレータの削除 (Decorator Deletion)
EHD	例外ハンドラの削除 (Exception Handler Deletion)
EXS	例外の握りつぶし (Exception Swallowing)
IHD	変数隠蔽の削除 (Hiding Variable Deletion)
IOD	オーバーライドされたメソッドの削除 (Overriding Method Deletion)
IOP	オーバーライドされたメソッド呼び出し位置の変更 (Overridden Method Calling Position Change)
LCR	論理結合子の置換 (Logical Connector Replacement)

次ページに続く

演算子	内容
LOD	論理演算子の削除 (Logical Operator Deletion)
LOR	論理演算子の置換 (Logical Operator Replacement)
ROR	関係演算子の置換 (Relational Operator Replacement)
SCD	super 呼び出しの削除 (Super Calling Deletion)
SCI	super 呼び出しの挿入 (Super Calling Insertion)
SIR	スライスのインデックス削除 (Slice Index Removal)
QGD	量子ゲートの削除 (Quantum Gate Deletion)
QGR	量子ゲートの置換 (Quantum Gate Replacement)
QMD	量子測定の削除 (Quantum Measurement Deletion)

ミューテーション演算子の違いが修正成功率に与える影響を調査するため，利用するミューテーション演算子が異なる 3 つの条件でミューテーションを行った．

1. 全てのミューテーション演算子を利用する
2. 量子ミューテーション演算子のみを利用する
3. 古典ミューテーション演算子のみを利用する

それぞれの条件下で，ミュータントを 1 つ以上生成できたプログラム数と，生成されたミュータントの総数を 表 2 に示す．量子ミューテーション演算子のみでミューテーションを行った場合に，ミュータントが 1 つも生成されなかったプログラムが 4 件あった．これらのプログラムは，いずれもソースコード内に量子ゲートが存在しなかった．

表 2: ミューテーションの結果

ミューテーション演算子	ミューテーション可能なプログラム数	ミュータントの個数
全て (量子 + 古典)	18	766
量子のみ	14	574
古典のみ	18	74

4.4 プロンプト構成

量子プログラムに対する APR において，プロンプトに含める情報が修復性能に与える影響を調査するため，本研究では以下の 8 種類のプロンプト構成を設計する (図 1 参照).

- ***S***：静的情報のみ.
- ***S+D***：静的情報および動的情報.
- ***S+M*(全ての演算子)**：静的情報およびミューテーション解析結果．全てのミューテーション演算子を利用.
- ***S+M*(量子演算子のみ)**：静的情報およびミューテーション解析結果．量子ミューテーション演算子のみを利用.
- ***S+M*(古典演算子のみ)**：静的情報およびミューテーション解析結果．古典ミューテーション演算子のみを利用.
- ***S+D+M*(全ての演算子)**：静的情報，動的情報，およびミューテーション解析結果．全てのミューテーション演算子を利用.
- ***S+D+M*(量子演算子のみ)**：静的情報，動的情報，およびミューテーション解析結果．量子ミューテーション演算子のみを利用.
- ***S+D+M*(古典演算子のみ)**：静的情報，動的情報，およびミューテーション解析結果．古典ミューテーション演算子のみを利用.

プロンプトに含まれる情報量は $S < S{+}D < S{+}M < S{+}D{+}M$ の順に増加し，この順序は，各情報を取得するために必要な労力の増加とも対応している．ミューテーション解析結果が含まれる $S{+}M$ と $S{+}D{+}M$ については，ミューテーション時に利用する演算子の違いで 3 種類の異なるプロンプト構成を設計した．それぞれの情報がどのように取得されるかについては，4.4.1 節，4.4.2 節，4.4.3 節で説明する.

システムプロンプトは，8 種類すべてのプロンプト構成で共通して同一のものを用いた．このシステムプロンプトでは，プロンプトに含まれる各情報の解釈方法および利用方法に関する詳細な指示に加え，厳密に従うべき出力形式が定義されている．実験で使用したシステムプロンプトの全文を付録に示す．本研究で使用したすべてのプロンプトは，レプリケーションパッケージにて公開している [38].

4.4.1 静的情報

本研究では，(1)Bugs4Q から取得したバグを含むコード，(2) 各バグのソース URL から取得したバグの説明，および (3) ソース URL に記載されている，プログラムが本来満たすべき正しい振る舞いを記述した期待される挙動の 3 種類の静的情報を収集した．(2) および (3) の情報は，2 名の調査者が各バグのソース URL を確認し，必要な情報を抽出することで手動で収集した．抽出内容に不一致が生じた場合には，協議を行い合意に達することで解決した．

4.4.2 動的情報

動的情報は，バグを含むプログラムの実行結果から構成され，エラーメッセージおよびそれに付随するスタックトレースを含む．プログラムが構文エラーなどにより実行できない場合には，エラーメッセージを含めている．実行は成功するものの，出力が期待される値と異なる場合には，テスト結果を含める．各プログラムにおけるバグの種類は，バグを含むコードおよび対応するテストコードの双方を実行することで判定した．バグの種類は，以下の 2 種類である．

1. **TE 型**: Qiskit ライブラリによって例外が送出される場合のように，プログラムの実行自体に失敗する例外発生 (*Throw Exception*) 型
2. **WO 型**: プログラムの実行は成功するものの，測定された量子状態が期待される状態と異なる場合に AssertionError が発生してテストで失敗する場合のような，誤った出力 (*Wrong Output*) 型

4.4.3 ミューテーション解析結果

本研究では，バグを含む各量子プログラムに対して QMutPy [13] を適用し，ミューテーション解析結果を取得した．それぞれの量子プログラムに対して複数のミュータントが生成される．生成されたすべてのミュータントに対して，元のプログラムに対応するテストスイートを実行し，その実行結果を判定する．本研究では，これらの一連の結果をミューテーション解析結果と呼び，各ミューテーション操作について以下の情報を含むものとする．

- *line_number:* ミューテーション演算が適用された行番号.
- *mutation_operator:* プログラムに適用されたミューテーション演算子の名称.

- *exception_traceback:* テスト実行が失敗した際にミュータントに対して出力されるトレースバック.
- *status:* ミューテーションテストの結果であり，以下のいずれかに分類される.
  - *killed*：少なくとも 1 つのテストにおいて実行結果が変化した場合.
  - *survived*：テスト結果に変化が見られない場合.
  - *incompetent*：実行不可能なミュータント (例：コンパイルエラー，クラッシュ).
  - *time_out*：実行が時間制限を超過した場合 (例：無限ループ).

4.5 大規模言語モデル

本研究では，OpenAI API を通じて最先端の大規模言語モデルである GPT-5 を，デフォルト設定のまま使用した．本モデルは，実験実施時点において利用可能であったモデルの中で，最新の推論モデルであったため選択した．LLM は非決定な性質を持ち，同一の入力に対して異なる出力を生成する可能性があるため，各プロンプト構成について 5 回ずつ出力を生成した．その結果，合計 680 件の修正コードおよびそれに対応する説明を取得した．それぞれのプロンプト構成で取得できた修正コードおよび説明の件数を 表 3 に示す.

表 3: 各プロンプト構成において取得できた修正コードおよびそれに対応する説明の件数

プロンプト構成	取得できた修正コードおよび説明の件数
S	90
S+D	90
S+M(全ての演算子)	90
S+M(量子演算子のみ)	70
S+M(古典演算子のみ)	90
S+D+M(全ての演算子)	90
S+D+M(量子演算子のみ)	70
S+D+M(古典演算子のみ)	90

4.6 評価指標

本研究では，量子プログラムに対する自動修復手法を評価するにあたり，修正が正しく行われたかどうかに加えて，その修正過程がどの程度説明可能であるかに重きを置く．そのため，修正成功率および説明性という 2 つの評価指標を用いる．

4.6.1 修正成功率

本研究では，生成された修正コードが Bugs4Q に含まれるすべてのテストを通過した場合に，その修復を成功とみなす．修正成功率は，対象のプログラムのうち，5 回の修復試行の少なくとも 1 回で修復に成功したプログラムの割合として定義する．

4.6.2 説明性

LLM が生成する説明には，バグの原因，修正箇所，および修正の根拠が含まれうる．これらに関する説明を評価するため，本研究では，パッチ説明に関する先行研究で提案されている 3 つの核となる要素 [35] を採用する．

- *Position*：バグが発生している箇所．
- *Cause*：バグが発生した理由．
- *Change*：バグがどのように修正されたか．

説明の品質は，先行研究で提案されている以下の 3 つの基準 [37, 36] に基づいて評価する．

- *Correctness:* 説明内容が，ベンチマークに含まれている正解の修正済みコードと照らして正確であるか．
- *Completeness:* 説明が，生成された修正パッチに含まれるすべての変更を十分に記述しているか．
- *Complexity:* 説明に，不要に複雑な記述が含まれていないか．

3 つの説明要素 (Position, Cause, Change) それぞれについて，3 つの評価基準 (Correctness, Completeness，Complexity) を満たしているかどうかを，二値判定 (はい/いいえ) により評価する．LLM が生成した説明に対して **Position** 要素を評価する際には，1) 説明は，バグの発生箇所を正解の修正済みコードに対して**正確に (Correctly)** 説明しているか，2) 説明は**十分に**

網羅的 (Complete) であり，修復のために変更されたすべての箇所に言及しているか，3) バグの位置を説明するにあたり，不要な**複雑さ (Complexity)** を含んでいないか，の 3 点を判定する．生成された修正コードにおいてプログラムのどの部分が修正されたか確認し，もし 1 箇所でも言及されていない箇所があれば，Position の Completeness については「いいえ」とする．**Cause** 要素を評価する際は，1) 説明は，バグの発生理由を**正確に (Correctly)** 説明しているか，2) 説明は**十分に網羅的 (Complete)** であるか，3) バグの発生理由を説明するにあたり，不要な**複雑さ (Complexity)** を含んでいないか，の 3 点を判定する．**Change** 要素を評価する際は，1) 説明は，バグの修正内容を，正解の修正済みコードに対して**正確に (Correctly)** 説明しているか，2) 説明は**十分に網羅的 (Complete)** であり，修復のために行ったすべての変更に言及しているか，3) 修正内容を説明するにあたり，不要な**複雑さ (Complexity)** を含んでいないか，の 3 点を判定する．生成された修正コードで行われたプログラムの変更を全て確認し，修正箇所については全て言及されているが，変更内容について 1 つでも言及がない場合，Position の Completeness は「はい」で，Change の Completeness は「いいえ」と判定する．

以上の評価手順により，パッチ説明の 3 要素 (Position，Cause，Change) それぞれについて 3 つの評価基準を適用するため，LLM が生成した各説明に対して合計 9 件 (3 要素 × 3 基準) の二値評価が行われる．ミューテーション解析結果が説明性に与える影響を調査するため，*S*, *S+D*, *S+M*(全ての演算子), *S+D+M*(全ての演算子) の 4 種類のプロンプト構成で比較する．各プロンプト構成につき 5 件の修正パッチが生成されるが，本研究ではそのうち最初に生成された修正パッチのみを評価対象とし，18 件のプログラム × 4 種類のプロンプト構成で合計 72 件の説明を評価した．各説明の評価は 2 名の評価者が独立に実施し，評価結果に不一致が生じた場合には，協議を通じて合意を形成した．

4.7 実験環境

実機の量子ハードウェアは利用可能性が限られており，またノイズの影響が大きいため，すべての実験は古典計算機上で実施した．Qiskit で記述されたプログラムは，古典計算機上のシミュレータとして実行可能である．使用した Python のバージョンは 3.9 である．使用した Qiskit 関連コンポーネントのバージョンは以下のとおりである．

- qiskit-aer 0.10.0

- qiskit-aqua 0.9.5
- qiskit-ignis 0.7.1
- qiskit-terra 0.20.0

さらに詳細なバージョン情報は，レプリケーションパッケージ [38] に記載している．

5　実験結果

本節では，提案手法の有効性を検証するために実施した実験結果を示す．本研究では，ミューテーション解析結果およびミューテーション演算子の違いが，量子プログラムの修正成功率および説明性に与える影響を分析することを目的とする．具体的には，RQ1 ではミューテーション解析結果の有無が修正成功率に与える影響を，RQ2 ではミューテーション演算子の違いが修正成功率に与える影響を評価する．さらに，RQ3 ではミューテーション解析結果が生成される説明の品質に与える影響について分析する．

5.1　RQ1: ミューテーション解析結果は量子プログラム APR の修正成功率にどのような影響を与えるのか

表 4 は，各プロンプト構成およびバグ種別ごとの修正成功率を示している．“Total” 列は 18 件すべてのプログラムに対する修正成功率を表し，“TE” 列および “WO” 列は，それぞれ各バグ種別 (TE：8 件，WO：10 件) に対する修正成功率を示している．18 件のプログラムのうち，17 件は少なくとも 1 つのプロンプト構成によって修復可能であった．

ミューテーション解析結果は，動的情報と組み合わせて用いることで最も高い効果を発揮する．すべてのプロンプト構成の中で，*S+D+M* は全体で 94.4% という最も高い修正成功率を達成した．WO バグのみを対象とした場合，*S+D+M* はすべてのプログラムを修復することに成功した．一方で，*S+M* の全体の修正成功率は，*S+D* よりも低かった．ミューテーション解析にはテストの実行が必要であるため，*M* が利用可能な場合には *D* も同時に利用可能であ

表 4: プロンプト構成およびバグ種別ごとの修正成功率．“WO” は Wrong Output，“TE” は Throw Exception を表す．各列における最大値を太字で強調している．

プロンプト構成	Total [%]	WO [%]	TE [%]
S	77.8	70.0	**87.5**
S+D	88.9	90.0	**87.5**
S+M	83.3	80.0	**87.5**
S+D+M	**94.4**	**100.0**	**87.5**

ることを意味する．したがって，*S+M* の修正成功率が低いことは問題にはならず，そのような場合には容易に *S+D+M* を構成できる．対照的に，TE バグについては，修復に成功したプログラムの集合 (8 件中 7 件) が，すべてのプロンプト構成で同一であった．これは，本カテゴリに属するバグを含むコードに対しては，動的情報やミューテーション解析結果が修復性能に影響を与えないことを示している．WO バグでは，プログラムはクラッシュせずに実行が完了するため，ミューテーション解析結果のような実行時情報が修復に有用である．一方，TE バグでは，プログラムはエラーによって停止するため，バグの存在が明確であり，静的情報のみで十分である．

Answer to RQ1: 動的情報とミューテーション解析結果を組み合わせることが最も効果的であり，全体の修正成功率は 94.4% に達した．また，Wrong Output 型のバグに対しては，修正成功率 100% を達成した．

5.2 RQ2: ミューテーション演算子の違いは量子プログラム APR の修正成功率にどのような影響を与えるのか

表 5 は，各プロンプト構成およびミューテーション演算子ごとの修正成功率を示している．

全ての演算子を利用した場合のミューテーション解析結果が，最も修正成功率を高める． *S+M*，*S+D+M* いずれのプロンプト構成の場合でも，全てのミューテーション演算子を利用

表 5: プロンプト構成およびミューテーション演算子ごとの修正成功率.

プロンプト	ミューテーション演算子	修正成功率 [%]
S+M	全て (量子 + 古典)	**83.3**
S+M	量子のみ	78.6
S+M	古典のみ	83.3
S+D+M	全て (量子 + 古典)	**94.4**
S+D+M	量子のみ	85.7
S+D+M	古典のみ	77.8

した場合が最も修正成功率が高いことがわかる．バグ種別に観察すると，TE バグを含むプログラムは全部で 8 件あるが，このうち 1 件のプログラムにおいて，量子ミューテーション演算子のみでは 1 つもミュータントが生成されなかった．このプログラムを除くと，TE バグに関する結果は，ミューテーション演算子を削減しても全く変わらなかった．これは，TE バグの修正において，ミューテーション演算子の違いが修正成功率に影響しないことを示唆している．一方，WO バグを含むプログラムは全部で 10 件あり，このうち 3 件のプログラムにおいて，量子ミューテーション演算子のみではミュータントが 1 つも生成されなかった．この 3 件のプログラムを除いた残りの 7 件のプログラムのうち，ミューテーション演算子を削減することによって，修正の成否に影響があったプログラムは 4 件ある．このうち 3 件のプログラムでは，ミューテーション演算子をどちらか一方にすると，修正に失敗するようになった．反対に，ミューテーション演算子を削減することで，修正可能になった事例も 1 件存在した．全てのミューテーション演算子を利用した場合に *S+M* プロンプトを利用すると修正できなかったプログラムが，量子ミューテーション演算子のみでミューテーションした場合に修正できた．この結果から，WO バグの修正においては，量子ミューテーション演算子の方が修正成功率の向上にわずかに寄与しうるといえる．

Answer to RQ2: 全てのミューテーション演算子を利用することが，量子プログラムの修正成功率の向上に効果的である．

5.3 RQ3: ミューテーション解析結果は量子プログラム APR の説明性にどのような影響を与えるのか

表 6 は，各プロンプト構成において，18 件のプログラムのうち各評価基準を満たしたプログラム数を示している．2 名の評価者の一致率は 79.2%（648 件中 513 件＝ 72 件の説明 × 9 項目の評価）であり，Cohen の κ [39] は 0.48 であった．この値は，中程度の一致を示すものと解釈される [40]．また，本結果は，バグ欠陥限局に関する LLM 生成説明を評価した先行研究で報告されている $\kappa = 0.55$ [41] と同程度である．

表 6 における太字の値から，*S+D+M* は 9 項目中 6 項目で最良の評価を達成しており，他のプロンプト構成と比較して，より良質な説明を生成できていることが分かる．特に Position 要素に関しては，*S+D+M* がすべての評価基準で最良の評価を達成しており，動的情報および

表 6: 各プロンプト構成における説明性．↑（↓）は，値が高い（低い）ほど良いことを示す．各行における最良の値は太字で強調している．

評価基準	要素	*S*	*S+D*	*S+M*	*S+D+M*
Correctness (↑)	Position	12	12	13	**14**
	Cause	**14**	13	12	12
	Change	7	**9**	5	7
Completeness (↑)	Position	13	**15**	12	**15**
	Cause	**18**	15	16	15
	Change	13	**16**	13	**16**
Complexity (↓)	Position	2	2	**1**	**1**
	Cause	2	2	**1**	**1**
	Change	8	7	8	**6**

ミューテーション解析結果を活用することは，バグ位置に関する説明において高い有効性を示している．一方で，*S+D+M* は Cause 要素に対しては相対的に有効性が低い．この要素については，Correctness および Completeness の両基準において，*S* のプロンプト構成が最良の評価を達成した．その理由の一つとして，修復の原因に関する情報の多くは，静的情報の中に既に含まれていることが考えられる．ミューテーション解析結果は，バグの原因を説明するよりも，詳細な位置情報に関する説明においてより効果的である．

Answer to RQ3: ミューテーション解析結果を動的情報と組み合わせることで，位置に関する記述がより正確かつ網羅的で簡潔になり，9 項目中 6 項目で最良の評価を達成した．一方で，バグの原因に関する説明性に対する影響は限定的である．

6　考察

6.1　プロンプト構成間における修復成功プログラムの差異

RQ1 において，動的情報およびミューテーション解析結果の有効性が示されたことから，本節では，これらの要素から特に恩恵を受けるプログラムを詳細に分析する．図 2 は，各プロンプト構成によって修復されたプログラム数を示している．すべてのプロンプト構成において，18 件中 14 件のプログラムが修復されている一方で，*S+D+M* 構成のみが 17 件のプログラムの修復に成功している．これは，*S+D+M* 構成が，修復に成功したすべてのプログラムを包含しており，動的情報およびミューテーション解析結果が修復性能を阻害していないことを示している．以上の結果から，ミューテーション解析は，実世界のバグを含む量子プログラムに対する LLM ベース APR の有効性を向上させる具体的な利点を提供することが示された．

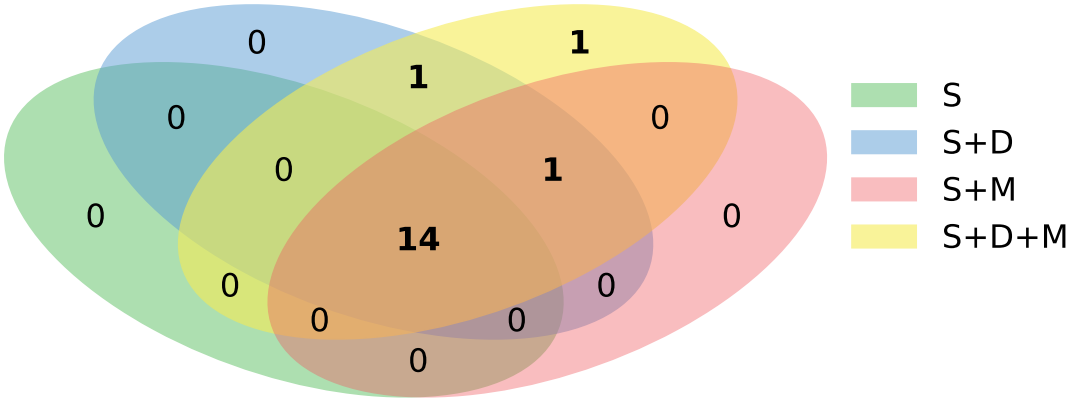

図 2: プロンプト構成別の修復成功プログラム数．

6.2　ミューテーション解析と修復結果の関係に関する事例分析

ここでは，ミューテーション解析結果が修復の成否や修正方針の決定にどのように寄与したのかを明らかにするため，定性的な分析を行う．

6.2.1 制御付きゲート置換における wire-mapping バグ

本節では，ミューテーション解析結果をプロンプトに含めることで修正に成功した量子プログラムの一例として，制御付きゲートを DAG 上で置換する際に量子ビットの対応関係が保持されないバグを取り上げる．本プログラムは，*S+D+M* 構成においてのみ修復に成功した事例である．

プログラムリスト 1 は，制御付き GHZ ゲートを回路に追加した後，DAG 上で制御を除去した等価なゲートへ置換する処理の抜粋である．ここでは，制御量子ビットとターゲット量子ビットの順序を，意図的に量子レジスタの順序と一致しない形で指定している．

プログラムリスト 1: 制御付きゲート置換を行うコードの修正例

```
  cghz = ghz_circuit.to_gate().control(1)
  circuit.append(cghz,[3,1,0,2])
  dag=circuit_to_dag(circuit)

  unctl_dag = DAGCircuit()
- for qreg in dag.qregs.values():
-     unctl_dag.add_qreg(qreg)
+ tmp_qreg=QuantumRegister(len(node.qargs),'q')
+ unctl_dag.add_qreg(tmp_qreg)
+ target_qargs=[unctl_dag.qubits[i]
+     for i in range(node.op.num_ctrl_qubits,
+     len(node.qargs))]
  unctl_dag.apply_operation_back(node.op.base_gate,
-     node.qargs[node.op.num_ctrl_qubits:],
+     target_qargs,
      node.op.params)
  dag.substitute_node_with_dag(node, unctl_dag)
```

プログラムリスト 2: 制御付きゲート置換を行うコードの修正時に生成された説明

```
The wire order was being shuffled because the replacement DAG was built from the
full circuit's registers, which do not match the node's qarg order.
DAGCircuit.substitute_node_with_dag maps wires positionally, so the replacement
DAG must list qubits in the exact order of the node's qargs.
The fix constructs a fresh temporary DAG with a new QuantumRegister sized to
len(node.qargs), preserving their order, and applies the base gate to the target
subset. This prevents wire permutation and yields the expected layout.
```

本来，この置換処理では，制御量子ビットの量子状態は変化せず，残りのターゲット量子ビットに対して GHZ ゲートが適用されることが期待される．しかし実際には，`substitute_node_with_dag`(17 行目) による置換後の回路において量子ビットの対応関係が入れ替わり，GHZ ゲートが本来とは異なる量子ビットに作用する回路が生成される．この結果，見た目上は同様の回路構造を持つにもかかわらず，意味的には異なる量子プログラムとなる．

このようなバグは，コードの静的構造からは DAG の構築やゲート置換処理が形式的には正しく記述されているように見える一方で，置換に伴う量子ビット対応関係の変化がコード上に明示的に表れないため，原因箇所の特定が困難である．

ミューテーション解析結果の観察

ミューテーション解析ログを確認すると，大部分のミュータントはテストを通過（*survived*）しており，テスト失敗（*killed*）は限られた箇所に集中して観測された．このことは，本不具合の影響がプログラム全体に及んでいるのではなく，特定の処理領域に局在していることを示唆している．

まず，置換用 DAG を生成する処理に対する**古典**ミューテーションでは，プログラムリスト 1 の `unctl_dag = DAGCircuit()`（5 行目）に関わる初期化やスコープの変更により，`unctl_dag` が代入されない実行経路が生じ，その後 `unctl_dag` を参照した時点で `UnboundLocalError` が発生したことがテストにより検出された．これは，本コードが `unctl_dag` の生成および代入が成功することを暗黙の前提として実装されており，その前提が崩れると直ちに破綻することを示している．また，制御付きゲート生成 `ghz_circuit.to_gate().control(1)`（1 行目）に関連する**量子**ミューテーションの一部では，制御付きゲートを内部的に分解・変換する処理において例外が発生し，テストが失敗するケースが観測された．

これらは，制御付きゲート生成およびそれに続く DAG 変換・置換処理の一部が，特定の内部的前提条件に依存して実装されており，その前提が満たされない場合にテストに失敗する経路が存在することを示唆する．一方で，量子ゲートの種類変更や，古典的な条件式・演算子に対する多くのミューテーションはテストを通過しており，本バグが特定の変換・置換処理周辺に限定された問題であることが確認された．

修正に成功した理由の推測

本事例において修正に成功した要因の一つとして，**量子ミューテーション演算子と古典**

ミューテーション演算子の両方を併用した点が挙げられる．実際に生成された説明 (プログラムリスト 2) では，「置換用 DAG が回路全体の量子レジスタに基づいて構築されており，置換対象ノードの `qargs` の順序と一致していないため，`substitute_node_with_dag` の位置ベースの対応付けによって wire の順序が入れ替わる」ことが不具合の原因として明示されている．この説明は，DAG 置換時における量子ビット順序という構造的要因に原因を帰着させており，量子ミューテーションによって「量子演算自体は本質的な原因ではない」ことが示唆され，古典ミューテーションによって「DAG 構築や置換処理が前提条件に依存している」ことが顕在化した結果として，このような特定に至った可能性がある．

6.2.2 量子回路の分解結果を返さない古典制御に関するバグ

本節では，量子回路の意味的誤りではなく，回路分解結果を関数の戻り値として返していないというプログラムの古典的な制御に起因するバグについて説明する．こちらは，*S+M* プロンプトにおいて，量子ミューテーション演算子が修正要因の特定に繋がった事例である．

プログラムリスト 3 にこのバグの修正例を示す．修正前のコードでは `decompose()` の結果を標準出力に表示するのみで，関数の戻り値としては分解前の回路を返していた．修正後では，`decompose()` が返す回路オブジェクトをそのまま戻り値として返すことで，テストが期待する回路表現と一致させている．これは，プログラムリスト 4 で説明されている内容と一致する．

プログラムリスト 3: decompose 結果を戻り値として返す修正例

```
  def main():
      qc = QuantumCircuit(1)
      qc.u1(0.24, 0)
-     print(qc.decompose())
-     return qc
+     return qc.decompose()

  if __name__ == "__main__":
-     main()
+     print(main())
```

プログラムリスト 4: decompose 結果を戻り値として返す修正時に生成された説明

```
Wrapped the circuit creation in a main() function and returned the decomposed
circuit so that the U1 gate is shown as its U3 decomposition, aligning with the
expected behavior and the test harness that calls main().
```

本バグは量子回路の意味的な誤りではなく，プログラムの制御フローに起因するバグである．`QuantumCircuit.decompose()` は分解後の新しい回路オブジェクトを返す API であるが，元のコードではその戻り値を無視し，分解前の回路を関数 `main()` の戻り値として返していた．その結果，U1 ゲートの分解自体は正しく行われているにもかかわらず，テストでは分解後の回路が観測されず，失敗していた．

ミューテーション解析結果の観察

量子ミューテーション演算子のみを用いた場合のログを分析すると，ミューテーションはすべてプログラムリスト 3 における 3 行目に適用されていることがわかった．`QGR`(量子ゲート置換) によってこの行に変更が加えられた多くのミュータントは，回路実行以前の段階で `Invalid bit index: '0.24'` という例外を発生させ，*killed* と判定されている．これは，量子ゲートを差し替えた結果，ゲート引数の順序や型が崩れ，量子的な意味の変化を観測する以前に，古典的な API レベルのエラーが発生していることを示している．一方で，一部のミュータントは *survived* や *timeout* となっており，量子ゲートの置換自体が必ずしもテスト結果に影響を与えていないと分かる．

修正に成功した理由の推測

S+M プロンプトにおいて量子ミューテーション演算子のみを用いた場合に修正が成功したのは，ミューテーションログが否定的な手がかりとして機能したためであると考えられる．量子ゲートを変更しても，意味的な差分を得る前に例外が発生する，あるいはテスト結果に影響しないミュータントが多く観測されたことで，量子回路そのものが修正対象ではないと推測しやすくなった．その結果，LLM の推論は量子演算の修正ではなく，回路の生成および値の返し方といった古典側の制御フローに集中し，`decompose()` の戻り値を返していないという本質的な誤りに到達できたと考えられる．一方で，*S+D+M* プロンプトでは，動的情報が加わることで，回路のどの状態が観測されているかがより明確になる．この場合は，ミューテーション演算子の種類に依らず，分解後の回路を返却するという修正に安定して到達できたと考えられる．以上より，本バグにおいては，ミューテーション情報単体よりも，動的情報が修正方向を決定する上で重要な役割を果たしていたといえる．

7　妥当性への脅威

7.1　構成妥当性

RQ1 と RQ2 では，すべてのテストを通過することを成功と定義したが，この定義は，修正内容が必ずしも開発者の意図した変更と完全に一致していることを保証するものではない．そのため，報告された修正成功率の妥当性に影響を及ぼす可能性がある．

7.2　内的妥当性

GPT-5 は非決定的な性質を持つため，同一のプロンプトであっても必ずしも同じ結果が得られるとは限らない．本研究では，各プロンプト構成につき 5 回の出力生成を行ったが，LLM 出力に内在するばらつきが，実験結果に影響を与えている可能性は否定できない．RQ3 においては，説明性の主観性を低減するため，2 名の評価者が手動で評価を行った．しかしながら，評価には一定の主観的判断が不可避であり，量子プログラミングに関する専門知識の制約により，誤分類が生じている可能性がある．

7.3　外的妥当性

本研究は，Qiskit のシミュレータ上で実行される量子プログラムを対象として実施しており，実機の量子計算機において同様の結果が得られるかどうかは明らかではない．また，本研究の調査対象は，単一のベンチマーク (Bugs4Q) および単一の量子フレームワーク (Qiskit) に限定されているため，より広範な実験対象 (他のバグベンチマーク，異なる量子フレームワーク，あるいは GPT-5 以外の LLM) を用いた包括的な検証が今後必要である．

8 おわりに

本研究では，実世界のバグを含むベンチマークを用い，量子プログラムに対する LLM ベース自動プログラム修復 (APR) において，ミューテーション解析結果をプロンプトに組み込むことの有効性を示した．実験の結果，静的情報，動的情報，およびミューテーション解析結果を含む *S+D+M* プロンプト構成が，全体で 94.4% という最も高い修正成功率を達成することが明らかとなった．また，量子ミューテーション演算子，もしくは古典ミューテーション演算子のみを利用するのではなく，全てのミューテーション演算子を利用する場合に，最も修正成功率が高いことが示された．さらに，*S+D+M* を用いて生成された説明は，バグ位置に関して正確かつ網羅的で簡潔な記述を提供しつつ，最も低い複雑度を示した．これらの結果は，ミューテーション解析結果を動的情報と組み合わせることで，修正成功率と LLM が生成する説明の品質の双方を向上できることを示している．本研究は，ミューテーション解析が量子プログラムに対する LLM ベース APR を改善するための有用な文脈情報となり得ることを示す，初の実証的証拠を提示した．この知見は，信頼性と説明可能性の双方を高める量子プログラム向け APR 手法の開発に向けた，新たな方向性を示すものである．今後の課題として，以下の 3 点を挙げる．第一に，追加的な文脈情報の活用による手法の堅牢化である．本手法の適用可能範囲は，ミューテーション解析が実行可能なプログラムに限定される．この制限を克服するため，Qiskit 等の公式ドキュメントから抽出した外部知識をプロンプトへ動的に統合し，ミューテーション解析を補完するアプローチを検討すべきである．第二に，開発者のドメイン知識ギャップを埋めるための支援への応用が考えられる．量子 OSS 開発者は物理学や量子計算の専門知識が不足している傾向にあることが指摘されている [4]．本研究で実現した説明生成能力を活用し，プログラムの修復過程で「なぜその修正が必要か」を提示することで，開発者が自身の知識不足を認識し，学習を促す仕組みの構築を目指す．第三に，実用的な開発ワークフローへの統合である．量子プログラムの開発やレビューには高度な専門性が必要であり，依然として困難が伴う．本手法を GitHub の Issue やプルリクエストのレビュー支援ツールとして実装・展開することで，実際の OSS 開発現場における説明の有用性を明らかにできると期待される．

謝辞

修士課程の道のりにおいては，たくさんの方にお世話になった．

まずは，弊研究室の肥後芳樹教授にお礼を申し上げる．肥後先生には，研究の中間報告会や日々の輪講において，研究の方向性や発表内容についてご助言をいただいた．特に発表資料に関しては，具体的かつ丁寧なアドバイスを数多くいただき，資料作成に対する自信を持てるようになったことは，自分にとって大きな財産である．さらに，国内外を問わず，何度も対外発表の機会を勧めていただいたことで，自分にとってハードルの高い目標にも踏み出すことができた．研究を通じて自身の能力を高め，国際会議で研究発表を行うことを目標として研究室に配属された当初の思いを形にできたことに深く感謝したい．

次に，指導教官である松下誠准教授にお礼を申し上げる．松下先生からは，週 1 回の研究ミーティングを通して，研究の進捗に関する継続的なご指導をいただいた．研究内容に限らず，スケジュール管理がうまくいかなかった際にも，一緒に状況を整理してくださり，どのように取り組むべきかについてご助言をいただいた．また，研究に行き詰まった際，常に本質を捉えたアドバイスをいただき，次に取るべき一手を明確にすることができた．なかなか納得のいく研究テーマに出会えず，思うように研究が進まない時期においても，ミーティングで相談に乗っていただいたことで，諦めずに修士論文に取り組めた．深く感謝申し上げたい．

次に，弊研究室の Raula Gaikovina Kula 教授にお礼を申し上げる．Raula 先生からは，研究の中間報告会や，輪講での発表に際して，研究テーマに関する非常に有益なご助言をいただいた．特に，輪講で紹介した論文に対するコメントを通して，論文の捉え方や研究の発展のさせ方について多くの示唆を得た．活発に議論をされる姿勢から，研究を楽しみながら深めることの大切さを学べたことに，深く感謝したい．

次に，弊研究室の Olivier Nourry 助教にお礼を申し上げる．Olivier 先生には，修士論文のテーマとの出会いのきっかけを与えていただいた．量子関連の研究に関心を持ちながらもテーマを模索していた時期に，量子分野の研究に取り組んでおられた石本優太氏をご紹介いただいた．また，英語論文の執筆に際しては，原稿を丁寧に添削していただき，根気強くご指導いただいた．その過程を通して，研究成果を正確かつ明確に伝えるためのテクニカルライティングについて学びを得られた．深く感謝申し上げる．

次に，九州大学の石本優太氏にお礼を申し上げる．石本さんには，本研究を進める上で，研

究テーマの設定から論文としてまとめる段階に至るまで，多岐にわたるご助言をいただいた．テーマを模索していた段階においては，量子分野に関する議論を通して，本研究テーマにつながる重要な着想を与えていただいた．また，研究をどのような順序で進めるべきか，何を明確にして進めるべきか，優先順位をどうすべきかについて，具体的かつ実践的なアドバイスをいただいた．さらに，論文執筆の過程では，内容や構成について議論を重ねながら，一緒に原稿を作り上げていただいた．共同で研究を進める過程を通して，論文としてまとめることを前提とした研究の進め方や，実験に着手する前段階で求められる見通しについて学ぶ機会を得られたことは，非常に貴重な経験であった．深く感謝申し上げたい．

次に，九州大学の亀井靖高教授にお礼を申し上げる．亀井先生には，英語論文の投稿に際して原稿の添削をしていただいた．さらに，本研究を進める過程において研究室訪問を快諾していただき，ミーティングを通じて直接議論する機会を設けていただいた．これらの機会を通して，研究内容を整理し，方向性を明確にした上で研究を円滑に進められたことに，深く感謝申し上げたい．

次に，九州大学の近藤将成助教にお礼を申し上げる．近藤先生には，論文投稿に際して原稿の添削をしていただいた．また，研究室訪問の際には，研究内容についてお話しする機会をいただき，多くの示唆を得られたことに深く感謝申し上げる．

そして，3 年間の研究生活を共にした研究室のメンバーに感謝したい．非常に優秀な仲間に囲まれて，良い刺激を受けながら楽しい学生生活を送ることができた．手探りで研究を進める中で，他のメンバーがどう工夫しているのかを知ることは非常に参考になった．専門知識の共有もさることながら，何気ない日常の会話をしたり，時に相談に乗ってもらったりと，様々な面でお世話になった．事務職員である軽部瑞穂氏，宮崎貴羅氏には，日々の手続や出張の手続きをサポートしてくださったことにも感謝申し上げる．

最後に，大学院まで通わせてくれた家族にお礼を言いたい．時に行き詰まり，悩むこともあったが，常に励まし続けてくれた家族の存在は何よりの支えだった．誰よりも私の可能性を信じてくれ，金銭面や精神面で有り余るほどのサポートをしてもらった．その支えがあったからこそ，自信を失わず，挑戦することを恐れない学生生活を送ることができた．家族の皆，ありがとう．

参考文献

[1] Michael A Nielsen and Isaac L Chuang. *Quantum computation and quantum information*. Cambridge university press, 2010.

[2] Juan Manuel Murillo, Jose Garcia-Alonso, Enrique Moguel, Johanna Barzen, Frank Leymann, Shaukat Ali, Tao Yue, Paolo Arcaini, Ricardo Pérez-Castillo, Ignacio García-Rodríguez de Guzmán, et al. Quantum software engineering: Roadmap and challenges ahead. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–48, 2025.

[3] Neilson Carlos Leite Ramalho, Higor Amario de Souza, and Marcos Lordello Chaim. Testing and debugging quantum programs: The road to 2030. *ACM Transactions on Software Engineering and Methodology*, 34(5):1–46, 2025.

[4] Ruslan Shaydulin, Caleb Thomas, and Paige Rodeghero. Making quantum computing open: Lessons from open source projects. In *Proceedings of the IEEE/ACM 42nd International Conference on Software Engineering Workshops*, page 451 – 455, 2020.

[5] Junjie Luo, Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. A comprehensive study of bug fixes in quantum programs. In *Proceedings of the 2022 IEEE International Conference on Software Analysis, Evolution and Reengineering*, pages 1239–1246. IEEE, 2022.

[6] Jake Zappin, Trevor Stalnaker, Oscar Chaparro, and Denys Poshyvanyk. When quantum meets classical: Characterizing hybrid quantum-classical issues discussed in developer forums. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 2931–2943, 2025.

[7] Yuechen Li, Hanyu Pei, Linzhi Huang, Beibei Yin, and Kai-Yuan Cai. Automatic repair of quantum programs via unitary operation. *ACM Transactions on Software Engineering and Methodology*, 33(6):1–43, 2024.

[8] Xiaoyu Guo, Jianjun Zhao, and Pengzhan Zhao. On repairing quantum programs using chatgpt. In *Proceedings of the 5th ACM/IEEE International Workshop on Quantum Software Engineering*, pages 9–16, 2024.

[9] Siwei Tan, Liqiang Lu, Debin Xiang, Tianyao Chu, Congliang Lang, Jintao Chen, Xing Hu, and Jianwei Yin. Hornbro: Homotopy-like method for automated quantum program repair. *Proceedings of the ACM on Software Engineering*, 2(FSE):734–756, 2025.

[10] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. Automated program repair. *Communications of the ACM*, 62(12):56–65, 2019.

[11] OpenAI. ChatGPT. `https://chat.openai.com/`, 2022. Accessed: 2025-11-04.

[12] Qihong Chen, Rúben Câmara, José Campos, André Souto, and Iftekhar Ahmed. The smelly eight: An empirical study on the prevalence of code smells in quantum computing. In *Proceedings of the IEEE/ACM 45th International Conference on Software Engineering*, pages 358–370, 2023.

[13] Daniel Fortunato, Jose Campos, and Rui Abreu. Mutation testing of quantum programs: A case study with qiskit. *IEEE Transactions on Quantum Engineering*, 3:1–17, 2022.

[14] Shaukat Ali, Paolo Arcaini, Xinyi Wang, and Tao Yue. Assessing the effectiveness of input and output coverage criteria for testing quantum programs. In *Proceedings of the 14th IEEE Conference on Software Testing, Verification and Validation*, pages 13–23, 2021.

[15] Eñaut Mendiluze, Shaukat Ali, Paolo Arcaini, and Tao Yue. Muskit: A mutation analysis tool for quantum software testing. In *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, pages 1266–1270, 2021.

[16] Eñaut Mendiluze Usandizaga, Shaukat Ali, Tao Yue, and Paolo Arcaini. Quantum circuit mutants: Empirical analysis and recommendations. *Empirical Software Engineering*, 30(3):100, 2025.

[17] Yuta Ishimoto, Masanari Kondo, Naoyasu Ubayashi, Yasutaka Kamei, Ryota Katsube, Naoto Sato, and Hideto Ogawa. Evaluating mutation-based fault localization for quantum programs. *arXiv preprint arXiv:2505.09059*, 2025.

[18] Pengzhan Zhao, Zhongtao Miao, Shuhan Lan, and Jianjun Zhao. Bugs4q: A benchmark of existing bugs to enable controlled testing and debugging studies for quantum programs. *Journal of Systems and Software*, 205:111805, 2023.

[19] Ali Javadi-Abhari, Matthew Treinish, Kevin Krsulich, Christopher J. Wood, Jake Lishman, Julien Gacon, Simon Martiel, Paul D. Nation, Lev S. Bishop, Andrew W. Cross, Blake R. Johnson, and Jay M. Gambetta. Quantum computing with Qiskit, 2024.

[20] OpenAI. Introducing gpt-5. `https://openai.com/index/introducing-gpt-5/`, 2025. Accessed: 2025-11-12.

[21] Charles H Bennett, François Bessette, Gilles Brassard, Louis Salvail, and John Smolin. Experimental quantum cryptography. *Journal of cryptology*, 5:3–28, 1992.

[22] Yudong Cao, Jonathan Romero, Jonathan P Olson, Matthias Degroote, Peter D Johnson, Mária Kieferová, Ian D Kivlichan, Tim Menke, Borja Peropadre, Nicolas PD Sawaya, et al. Quantum chemistry in the age of quantum computing. *Chemical reviews*, 119(19):10856–10915, 2019.

[23] Jacob Biamonte, Peter Wittek, Nicola Pancotti, Patrick Rebentrost, Nathan Wiebe, and Seth Lloyd. Quantum machine learning. *Nature*, 549(7671):195–202, 2017.

[24] Moses Openja, Mohammad Mehdi Morovati, Le An, Foutse Khomh, and Mouna Abidi. Technical debts and faults in open-source quantum software systems: An empirical study. *Journal of Systems and Software*, 193:111458, 2022.

[25] Yuta Ishimoto, Yuto Nakamura, Ryota Katsube, Naoto Sato, Hideto Ogawa, Masanari Kondo, Yasutaka Kamei, and Naoyasu Ubayashi. An empirical study on self-admitted technical debt in quantum software. In *Proceedings of the 31st Asia-Pacific Software Engineering Conference*, pages 41–50, 2024.

[26] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *IEEE Trans. Softw. Eng.*, 37(5):649 – 678, 2011.

[27] Eñaut Mendiluze Usandizaga, Shaukat Ali, Tao Yue, and Paolo Arcaini. Quantum circuit mutants: Empirical analysis and recommendations. *Empirical Software Engineering*, 30(4):100, 2025.

[28] Quanjun Zhang, Chunrong Fang, Yuxiang Ma, Weisong Sun, and Zhenyu Chen. A survey of learning-based automated program repair. *ACM Transactions on Software Engineering and Methodology*, 33(2):1–69, 2023.

[29] Chunqiu Steven Xia, Yuxiang Wei, and Lingming Zhang. Automated program repair

in the era of large pre-trained language models. In *Proceedings of the 45th International Conference on Software Engineering*, pages 1482–1494. IEEE, 2023.

[30] Matthew Jin, Syed Shahriar, Michele Tufano, Xin Shi, Shuai Lu, Neel Sundaresan, and Alexey Svyatkovskiy. Inferfix: End-to-end program repair with llms. In *Proceedings of the 31st ACM joint european software engineering conference and symposium on the foundations of software engineering*, pages 1646–1656, 2023.

[31] Chunqiu Steven Xia and Lingming Zhang. Automated program repair via conversation: Fixing 162 out of 337 bugs for $0.42 each using chatgpt. In *Proceedings of the 33rd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 819–831, 2024.

[32] Islem Bouzenia, Premkumar Devanbu, and Michael Pradel. Repairagent: An autonomous, llm-based agent for program repair. In *Proceedings of the IEEE/ACM 47th International Conference on Software Engineering*, pages 694–694, 2025.

[33] Ramtin Ehsani, Esteban Parra, Sonia Haiduc, and Preetha Chatterjee. Hierarchical knowledge injection for improving llm-based program repair, 2025.

[34] Google. Google best practice, 2025. Accessed: 2025-11-13.

[35] Jingjing Liang, Yaozong Hou, Shurui Zhou, Junjie Chen, Yingfei Xiong, and Gang Huang. How to explain a patch: An empirical study of patch explanations in open source projects. In *Proceedings of the 2019 IEEE 30th International Symposium on Software Reliability Engineering*, pages 58–69, 2019.

[36] Dominik Sobania, Alina Geiger, James Callan, Alexander Brownlee, Carol Hanna, Rebecca Moussa, Mar Zamorano López, Justyna Petke, and Federica Sarro. Evaluating explanations for software patches generated by large language models. In *Proceedings of the International Symposium on Search Based Software Engineering*, pages 147–152. Springer, 2023.

[37] Meike Nauta, Jan Trienes, Shreyasi Pathak, Elisa Nguyen, Michelle Peters, Yasmin Schmitt, Jörg Schlötterer, Maurice van Keulen, and Christin Seifert. From anecdotal evidence to quantitative evaluation methods: A systematic review on evaluating explainable ai. *ACM Comput. Surv.*, 55(13s):42, 2023.

[38] Chihiro Yoshida. Leveraging mutation analysis for llm-based automated quantum

program repair. https://doi.org/10.5281/zenodo.17626083, 2025.

[39] Jacob Cohen. A coefficient of agreement for nominal scales. *Educational and psychological measurement*, 20(1):37–46, 1960.

[40] J Richard Landis and Gary G Koch. The measurement of observer agreement for categorical data. *biometrics*, pages 159–174, 1977.

[41] Sungmin Kang, Gabin An, and Shin Yoo. A quantitative and qualitative evaluation of llm-based explainable fault localization. *Proceedings of the ACM on Software Engineering*, 1(FSE):1424–1446, 2024.

付録

プログラムリスト 5: システムプロンプト

```
# Task Description
As a debugger, you should return the fixed source code for the Buggy Code, together with a
    concise explanation of the fix, strictly following Output Format.
You may utilize all available information provided in the user prompt -- including the
    Buggy Code, Bug Description, Expected Behavior, Current Result, and Mutation Analysis
    Result --
to infer the intended behavior and determine the necessary code modification.

# Information Items
- Buggy Code: <description>
- Bug Description: <description>
- Expected Behavior: <description>
- Current Result: <description>
- Mutation Analysis Result: <description>

# Mutation Analysis Context
To gain a deeper understanding of the behavior of the Buggy Code, we generate *mutants* by
    applying a *mutation operator* that modifies exactly one statement in the Buggy Code.
Each mutant is executed against the same test suite used for the Buggy Code.
Multiple mutants can be generated from a single Buggy Code.

## Mutation Operator Types
<Mutation Operator's List>

## Mutation Test Status
The result of each mutant is categorized into one of four statuses:
<Mutation Test Status List>

## Mutation Result Format
The results of each mutation are presented as follows:
```yaml
<Each Mutation Result Format>
```

When multiple mutants are executed for a single Buggy Code, the mutation list contains
    multiple items.

Output Format
```

````
Please output in the following format.
---
fixed_code:
```python
{FIXED_CODE}
```
fix_description: {FIXED_DESCRIPTION}
---

Important:
- For `fixed_code`, output the full source code **exactly as-is**, preserving all line
    breaks and indentations.
- Do NOT escape newlines (no "\n").

Example:
---
fixed_code:
```
<Python Sample Code>
```
fix_description: <Fix Description Sample Text>
---
````