大規模言語モデルを活用した設計書からのソースコード生成

Source Code Generation from Design Documents Using Large Language Models

町　田　　凌

要　約　大規模言語モデルを用いたソースコード生成は，開発効率の向上が期待される一方で，生成するコードの規模が大きいほど，生成コードの正確性を保証する課題が大きくなる．筆者らが開発した設計書をインプットとするコード生成ツールは，複数エージェントの協調により設計情報に基づくコード生成・エラー修正を実現し，精度と効率の向上を図った．動作検証では，自律的なエラー修正が機能する一方で，テストコード生成には不備が確認された．今後はテスト精度の向上とLLMの出力の不確実性に対応する改良を重ね，実用的な完全自動化を目指す．

Abstract　The automatic code generation tool utilizing LLMs is expected to improve development efficiency, yet the larger the generated code becomes, the greater the challenge to ensure the accuracy of generated code might be. The code generation tool we developed, which uses design documents as input achieves enhanced precision and efficiency by coordinating multiple agents to generate and correct code based on design documents. Operational tests showed that autonomous error correction functions effectively, though issues were identified in test code generation. Moving forward, we aim to improve test accuracy, address uncertainties in LLM outputs, and work towards achieving practical, fully automated code generation.

1. はじめに

ソフトウエア開発におけるソースコード（以下，コード）の自動生成には，開発効率の向上やコスト削減に大きく貢献する可能性があることから，多くの研究や実用化が試みられている．BIPROGY株式会社（以下BIPROGY）でも，データベースのテーブル定義に基づいたDDL（Data Definition Language）やデータアクセスオブジェクト，画面項目定義に基づいた入力値のバリデーションチェックなどを対象に，テンプレートエンジンなどを用いて定型的なコードの自動生成を実施してきた[1]．

近年，注目を集めている大規模言語モデル（Large Language Model，以下LLM）は自然言語の理解と生成に優れ，人間と同等レベルの自然言語処理タスクを実行できる．プログラミングタスクにおいても，高い精度でコードを生成することができ，特に，ビジネスロジックのような非定型な部分の自動生成に多くの期待が寄せられている．BIPROGYでも，LLMを用いたコード自動生成ツールの活用に向け，各種検証を進めている．

一方で，LLMによるコード生成には課題がある．人間がコーディングする場合は，利用するモジュールの内容やモジュール間の依存関係を確認するために，コードベースを探索する．また，プロジェクトのコーディングスタイルに倣うために，既存のコードを理解する．そして，コーディング後は作成したコードの品質を確認するために，テストコードを作成してテストを行い，設計通りに実装できているか確認する．しかし，LLMはあくまでAIであり，コードベー

スに理解を深めることや，生成した実装の正確性を保証することを，人間のように自力では行えない．そこで，LLM の不確実性を克服し，期待した生成物が得られることを担保するための仕組みを LLM の外側に作り，生成されたコードの正確性を保証することとした．

現在，筆者らはこれらの課題を解決するアプローチを検討し，プロトタイプを開発している．最終的な目標は，ユーザが開発の上流工程で作成した成果物をツールに入力するだけで，設計情報に基づき正確性が担保されたコード一式が自動生成されることである．プロトタイプでは，ユーザが提供した設計書を基に，設計情報に対応するプロダクションコードおよびテストコードを自動生成する．内部には，クラスやメソッド単位で正確性を担保する仕組みを組み込んであり，生成されるコードの正確性向上に寄与している．

プロトタイプには，LLM と外部ツールを組み合わせて専門的なタスクを処理する「エージェント」と呼ばれる仕組みを搭載し，複数のエージェントが相互に連携して，コードを生成する．具体的には，設計書や既存のコードを基に実装コードの依存関係を特定するエージェントや，プロジェクト固有のコーディングスタイルを学習するためのサンプルコードを収集するエージェントが搭載されている．これらのエージェントにより，利用する情報を自動的に収集し，コードベースを理解したプログラマのようにコードを生成できる．また，コンパイラやテストツールを用いて自律的にエラーを検出し，その出力を基にコードを修正するエージェントも備わっている．このプロトタイプを用いることで，ユーザは，ビジネスロジックが適切に実装され，テストによって正確性が担保されたコードを，コーディングせずに得られるようになる．

本稿では，コード生成に LLM を適用するに至った背景や LLM の仕組み，その課題に触れた後，開発中のプロトタイプの現状と今後の開発方針について報告する．まず 2 章で本活動の背景や LLM の概要を説明し，3 章で LLM を活用したコード生成ツールの分類と BIPROGY のアプローチについて述べる．4 章ではコード生成の具体的な手法を紹介し，5 章でプロトタイプのアーキテクチャを詳しく解説する．6 章では動作検証の結果を報告し，7 章で総括と今後の展望について述べる．

2. 大規模言語モデルによる製作工程の効率化と課題

システムインテグレーションの現場では，開発 5 工程（基本設計，詳細設計，製作，結合テスト，総合テスト）の中でも，とりわけ「製作工程」が全体工数の約 3 割という大きな比重を占めている[2]．製作工程はプログラミングを中心とした作業であり，この工程の生産性向上は，プロジェクト全体の生産性を高めることになるため，注目を集めてきた．これまでは主に定型的な自動化手法を用いることで，製作工程の効率化を図ってきた．

近年注目を集めている LLM は，様々な自然言語タスクに適用されており，プログラミングタスクにおいても高精度なコード生成を実現できる．BIPROGY では，LLM を活用することで，これまで自動化が困難とされてきたビジネスロジックなどの非定型なコード生成が可能になると期待し，LLM を用いたコード自動生成ツールの有用性を検証している．

LLM の特徴の一つとして，文脈に基づく柔軟な応答が挙げられる．この特性が非定型なコードの生成を実現する原動力となる．LLM はユーザから与えられる「プロンプト」（入力や質問）を基に応答を生成する．さらに，プロンプトに知識情報を追加することで，LLM が事前学習では得られなかった情報を補完し，より高度な応答が得られる．この手法は「文脈内学習（In-Context Learning）」と呼ばれ，LLM があらかじめ学習した知識を補強し，特定の指示や状況

に即した回答を導出できる．プログラミングタスクにおいては，コードベースの情報を文脈内学習させることが鍵となる．

ただし，このような高い応用力にも限界がある．LLM は言語パターンを統計的に予測して応答を生成しているに過ぎず，その出力はしばしば曖昧さや不正確さを含む．この「正しく見えるが実は誤っている」回答を生成する現象は「ハルシネーション」と呼ばれる．特に，曖昧なプロンプトや不正確な情報を与えた場合に起こりやすい．そのため，LLM を活用する際には，正確な情報を明確なプロンプトで提示することが望まれる．それでもハルシネーションを完全に防ぐことは難しく，最終的な結果を人間の目視や自動テストによって確認するプロセスが不可欠である．

総じて，LLM はプログラミングタスクにおいて非常に有力なツールであり，今後さらなる活用が期待される．一方で，常に正確な応答を得られるわけではないことを理解しておかなければならない．LLM の技術的な限界を踏まえた上で，人間による検証を組み合わせるなど，LLM の可能性を最大限に引き出すための仕組みを構築することが重要である．

3. LLM を活用したコード生成の概要

現在，LLM を活用したコード生成ツールは広く開発され利用されており，プログラミングの効率を飛躍的に向上させる可能性を秘めている．本章では，これらのツールを分類別に考察し，大規模開発に適したものを選定する．

3.1 コード生成ツールの分類と考察

本節では，コード生成ツールを，その生成形態に着目して，統合開発環境（Integrated Development Environment，以下 IDE）に組み込まれた「IDE 一体型」，小規模なタスク単位のコード生成に特化した「AI エンジニア型」，より大規模なコードを一括して生成する「一括生成型」の 3 種類に分類して考察する．

一つ目の「IDE 一体型」ツールは，コーディング中の開発者をリアルタイムで支援することを目的としている．このタイプのツールは IDE に組み込まれており，数行レベルのコードを補完したり提案したりする．これにより，開発者はコードを記述しながら，生成されたコードを即座に確認および修正できる．例えば，GitHub Copilot[*1] や Cursor[*2] のようなツールは，コーディング中に適切なコードを提案し，開発者の入力を補助する．ただし，IDE 一体型ツールは基本的に部分的なコードしか生成せず，大規模なコードを一度に生成することは難しい．

二つ目の「AI エンジニア型」ツールは，ユーザとの対話を通して小規模なタスク単位のコードを生成することを目的としている．このタイプのツールは，チャット型インターフェースを用いてタスクの内容を理解し，技術的な知識がないユーザでも利用できるよう設計されている．例えば，Devin[*3] や Openhands[*4] などは，システム設計から実際のコード生成までの一部のエンジニアリングタスクを代替できる．また，このタイプのツールは，テスト機能や Web 検索機能を組み込み，外部リソースに自律的にアクセスして問題解決を支援する機能を持つこともある．ただし，生成されるコードの規模が IDE 一体型よりも大きいため，生成されたコードを確認する人間の負担が大きくなる．人間のレビューで起こりうる抜け漏れを防ぐため，テスト機能を活用して，与えたタスクの内容の通り正確に実装できているかを検証するなど，レビュー以外の仕組みでも品質を担保することが重要である．

三つ目の「一括生成型」ツールは，特に大規模な開発プロジェクトやシステムインテグレーション（SI）の下流工程を支援することを目的としている．これらのツールは設計書を基にしてコードを一括で生成し，ユーザとのやり取りを最小限に抑えながら，プロジェクト全体の効率を高めることができる．例えば，CodeAGI[*5] のようなツールは，数十ファイル以上の大規模なコードを自動的に生成でき，ウォーターフォール型の開発プロセスで特に有効である．

3. 2　大規模開発に適したコード生成ツールの実現に向けて

前節での分析と評価を踏まえ，大規模開発案件に適したコード生成ツールの開発を目指す筆者らは，一括生成型のコード生成アプローチを採用した．しかし，このタイプのツールでは，一度に生成されるコードが極めて多く，ユーザがコードをレビューし，その正確性を判断するのが非常に難しい．そのため，「AI エンジニア型」の説明で述べた自律的なテストのような，正確性を支える仕組みの構築がより重要になる．LLM が生成する不正確なコードに対処するため，自律的なテスト実行と修正機能をワークフローに組み込み，生成されたコードの正確性を向上させる仕組みを取り入れている．これにより，LLM による不確実性を克服しつつ，効率的かつ安全な大規模開発支援の実現を目指している．

4.　コード生成のアプローチ

本章では，設計書から正確で高品質なコードを生成するために筆者らが採用した具体的なアプローチを詳述する．

4. 1　エージェント

最初に，エージェントの概念とその役割について説明する．Schulhoff らは，エージェントを「生成 AI 自体の外部にあるシステムと連携し，ユーザの目的を達成するためにアクションを実行する生成 AI システム」と定義している[3]．つまり，エージェントは単にデータを生成するだけでなく，外部のリソースやツールと連携して，ユーザが望む結果を得るために自律的に行動するものである．

エージェントをツールに組み込むことで，以下のメリットが得られる．

- タスクの専門化：エージェントごとに特定の役割や専門知識を持たせることで，複雑なタスクを分割し，効率的に処理できる．
- 自律的な行動：エージェントが目的に応じたアクションを自律的に実行するため，ユーザによる介入を最小限に抑えられる．
- 外部との連携：外部のリソースやツールと連携することで，データの取得や処理，テストの実行など，多様な機能を統合的に活用できる．

エージェントは，アクションの決定や外部連携の際の入力項目の作成，外部から得たコンテンツの整形など，様々な処理を内部に持つ．エージェントの処理は非定型なものが多く，LLM を活用する場面が多くなる．そのため，エージェント内部では LLM が複数回呼び出される．エージェントの実装にあたっては，LLM の不確実性を鑑みると，エージェント単体での安定性を確保するためにプロンプトの設計や検証がより重要となる．

エージェントは専門性を持ち，他のエージェントと役割を分担するため，人間と似た性質が感じられることもある．また，複数のエージェントを協調動作させるマルチエージェントシステムの最終的な構成には，人間の思考プロセスや組織構造と似たものも存在する[4]．そのため，マルチエージェントシステムの開発にあたっては，実世界の事例が参考になる．しかし，LLM の出力は不確実なものであり，マルチエージェントシステムを実装する際は，人間の性質と似ているからと安易にエージェントを追加することは避けるべきである．プロトタイプを迅速に開発するためには，全体の安定性を意識しながら，新しく実装するエージェントが有効かどうかを十分に検証するべきである．

設計書を基にコードを書く開発者の思考プロセスを以下 1）～ 5）のように考察する．

1）開発者が物理設計書を基にコーディングする際，最初に確認するのは依存関係である．実装するメソッドから呼び出される予定のクラスについて確認し，開発に取り掛かる．
2）また，実装する処理のパターンと類似する既存の実装を確認しながら，コーディングする場合もある．これは，他の実装のコーディングスタイルに倣うためである．
3）コードベースから様々な実装をサンプルとして確認した開発者は，いよいよ実装に取り掛かる．
4）同時に，メソッドの実装が正しいものであることを機械的に確認できるよう，ユニットテストを実装する．
5）その後，実装した内容がテストを通過するか確認し，誤りがあった場合は修正する．

筆者らがプロトタイプで採用したマルチエージェントの構成も，最終的に人間がコーディングする際の思考プロセスと似たものとなった．その詳細については，5 章で述べる．

4.2　複数のエージェントを協調させるアプローチ

エージェント間の協調と効率的なタスク遂行を実現するためには，二つのアプローチが考えられる．一つ目はあらかじめ定型化されたワークフローに従ってエージェントを実行する方法であり，二つ目はエージェントが実行計画を立案し，各エージェントにタスクを動的に割り当てる Planner[5]エージェントを導入する方法である．

定型ワークフローでは，あらかじめ定義された手順に従いエージェントがタスクを遂行する．この方法では，4.1 節の 1）～ 5）で述べたコーディングタスクの各ステップを決められた順序で実行することで，エージェント間の連携が円滑になる．メリットとしては，エージェントの動作が予測でき調整が容易なことや，ワークフローに確実性があるため連携ミスを防げることなどが挙げられる．また，ワークフローが固定されているため開発とテストが容易になり，開発スピードが向上する．

Planner エージェントを用いるアプローチでは，LLM が全体のタスクフローを管理し，エージェントへの指示を管理する．Planner エージェントはタスクのスケジューリングやエージェント間のデータの受け渡し，進捗の監視などの役割を担う．これにより，タスクの柔軟な実行とエージェント間の連携強化を実現する．例えば，簡単なメソッドの実装であれば，時間のかかる情報収集系のエージェントをスキップし，直接コード生成用のエージェントを呼び出すなど，状況に応じて柔軟に対応できる．

筆者らはプロトタイプ開発にあたって，定型ワークフローを採用した．その理由は，プロトタイプ開発におけるプロジェクトの規模やタスクの複雑性が限定的であり，固定化された手順で十分な効率化が見込めることにあった．また，Planner エージェントは LLM を活用する特性上，その計画立案には不確実性が伴う．プロトタイプ開発では早期の実装が求められるため，この不確実性を排除し，個々のエージェントの性能向上に注力すべきと判断した．さらに，定型ワークフローを用いることで，ワークフローの設計とテストが容易になり，開発期間の短縮と正確性の向上に繋がると考えた．

4.3　プロジェクト固有のコードサンプルの利用

個々の開発プロジェクトにおいて，例外処理の方法，ログの出力形式，アノテーションの使用法などに関する固有の設計パターンを定義して，それに従って設計することや，固有のコーディングスタイルに準拠したコードを作成することは，開発生産性や作成物の保守性の観点から重要である．しかし，これらの知識を LLM に期待することはできない．また，コードの依存関係についても，現状のコードベースを確認する仕組みがなければ LLM が理解することはできない．

このギャップを埋めて高品質なコードを生成するためには，対象プロジェクトに固有のコードベースの中からコードサンプルを LLM に提供することが効果的である．しかし，コードベース全体をそのまま LLM に渡すと，情報量が多すぎてプロンプトが曖昧になったり，LLM が一度に処理できる情報量の上限（コンテキストサイズ）を超えてしまったりすることがある．それを防ぐため，関連性の高いコードサンプルを効率的に選び出して LLM に渡す仕組みを用意する．

プロトタイプの開発にあたって，プロジェクト固有のコードサンプルを効果的に活用するために，いくつかの機能を持つコード解析モジュールを実装した．このモジュールを利用することで，エージェントはコードベースから適切なコードサンプルを得られるようになる．このモジュールは構文解析，情報のマッピング，コードサンプルの抽出といった機能を備えている．エージェントに対して，コードベースに存在するクラスやメソッドの定義やファイルの場所，行番号の情報をマップ形式で提供することで，適切なコードサンプルの抽出を補助する．このモジュールの実装には，オープンソースのパーサジェネレータ「ANTLR」[*6] を利用している．

このコード解析モジュールは，4.1 節 1）～ 5）のコーディングの思考プロセスの各所で利用できる．コーディングに向けた情報を収集するステップでは，エージェントはコード解析モジュールが持つ情報を基に，コード生成に役立ちそうなコードサンプルを選び出す．このとき，コード解析モジュールは，クラスやメソッドの一般的な情報だけでなく，全体観を捉えさせるために，コードベースのディレクトリ構造を提供する．また，具体的な実装情報がコードベースの情報抽出に役立つため，タスクとの依存関係があると判明しているクラスやメソッドが存在する場合は，そのコード断片をエージェントに提供する．これにより，エージェントはコードベースの構造やタスクに関連する具体的な実装情報を基にして，より適切なコードサンプルを選び出せるようになる．こうして選ばれたコードサンプルはコード生成のためのエージェントに提供され，新しいコードを生成するための参考情報となる．このプロセスを通して，依存関係をよく理解して，プロジェクト固有のコーディングスタイルに従ったコードを生成することができる．

4.4　自律的なテストと修正

LLMを用いたコード生成では，生成されたコードの正確性を保証することが重要である．しかし，LLMが生成したコードには，コンパイルエラーやロジックの不整合が含まれることがある．これらの問題を人手で検証・修正することは時間と労力を要し，却って生産性を低下させる恐れがある．また，LLMに不正確なコードの有無をチェックさせても，確実に発見できる保証はない．

この課題を解決するために，プロトタイプ開発にあたってコード修正のためのエージェント（以下，コード修正エージェント）を実装した．コード修正エージェントは，コンパイラからのコンパイルエラーやテストツールからのアサーションエラーの出力を基に，コードを自律修正する．この仕組みにより，LLM特有の誤りを確実かつ効果的に検出・修正することができる．コード修正エージェントの処理について，図1にフローチャートを示し，以下1）～8）で解説する．

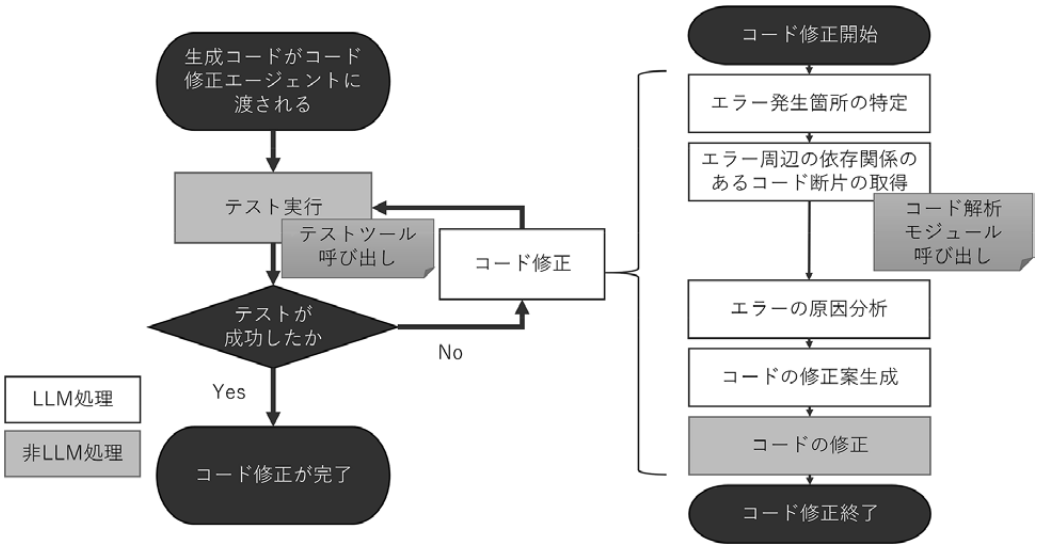

図1　コード修正エージェントの処理

1）テスト実行

コード修正エージェントは，まずテストツールを呼び出してテストを実行する．この際，コード生成用のエージェントが事前に生成したテストコードを使ってプロダクションコードをテストする．本プロトタイプでは，エージェントが呼び出すテストツールとして様々なツールを割り当てることができ，ビルドツールを割り当てることでコードのコンパイルエラーのみを検出することもできる．

2）テスト結果の成否

テスト結果を確認し，エラーの有無を判定する．エラーがない場合は，コードが正しく動作していると判断し，処理を終了する．

3）エラー発生箇所の特定

エラーが発生した場合は，テストツールが出力したログを解析し，エラーの発生箇所を特定する．

4）情報の収集

エラーの原因を詳しく調査するために，4.3節で述べたコード解析モジュールを利用し，エラー発生箇所のコードや依存関係の情報を収集する．

5）原因の分析

収集した情報を基に，エラーの原因を分析する．

6）コードの修正案の生成

エラーの原因に基づき，具体的な修正案を作成する．

7）コードの修正

修正案に従って，プロジェクト内の該当ファイルを修正する．

8）再テスト

修正後，再度テストを実行し，エラーが解消されたか確認する．エラーが残っている場合，このプロセスを繰り返す．

このように，コード修正エージェントは自律的にエラーを検出・修正する．これにより，開発者の手を煩わせることなく，コードの正確性を向上させることができる．

5. プロトタイプのアーキテクチャ

本章では，プロトタイプのアーキテクチャについて説明する．本プロトタイプはマルチエージェントシステムとして設計されたアプリケーションであり，サブルーチンとなる各エージェントがそれぞれの役割を担って連携することで，コード生成を実現する．プロトタイプのアーキテクチャを図2に示す．

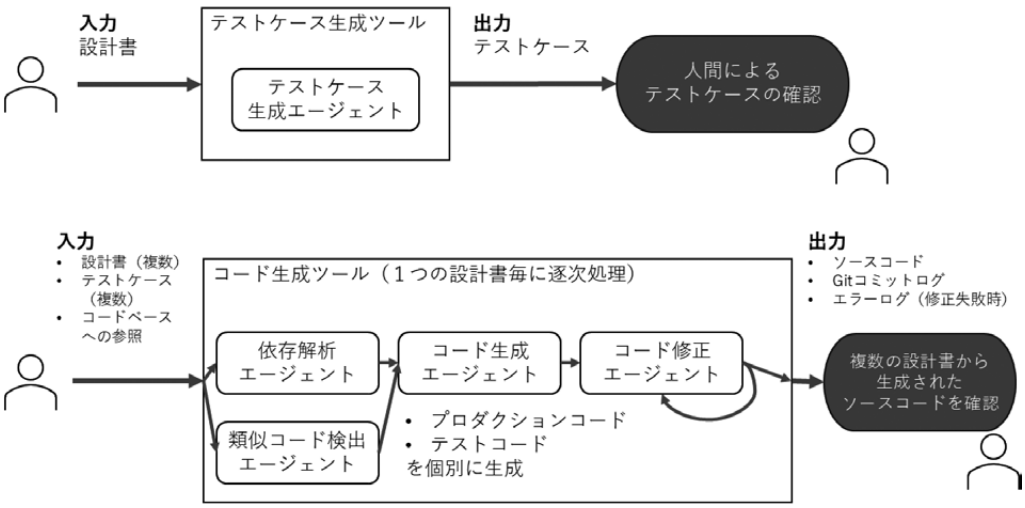

図2　プロトタイプのアーキテクチャ

プロトタイプは，設計書からテストケースを生成することに特化した「テストケース生成ツール」と，コード生成を担当する「コード生成ツール」の二つのアプリケーションで構成されている．前者はウォーターフォールの上流工程での利用を，後者は下流工程での利用を想定

している．テストケース生成とコード生成を分割している理由は，このプロトタイプは自動テストでコード品質を保証しており，テストケースの網羅性が重要だからである．現段階ではテストケースの網羅性を LLM 自身で判断することは避け，ユーザがテスト生成ツールを利用した後，生成されたテストケースをレビューして網羅性を確認し，適宜修正することを想定している．人の手を介すことで，テストの信頼性が高まる．

これらのアプリケーションは Python で実装されている．LLM に関連する処理については LLM フレームワークの LangChain[*7] を採用した．アプリケーション内部のエージェントは LangChain の Runnable クラスを継承して実装しており，エージェントはデータオブジェクトを受け渡すことで連携する．

テストケース生成ツールは，設計書からテストケースを生成するための一つのエージェントで構成される．ユーザが設計書を入力すると，その内容に対応するテストケースが生成される．ユーザは生成したテストケースが設計内容を網羅しているか確認し，適宜修正する．その後，生成されたテストケースをコード生成ツールに渡してコードを生成する．

コード生成ツールは，設計書とテストケース，コードベースへの参照を入力することで，コードベースへ生成コードの書き込みを行う．複数の設計書が入力された場合，コード生成ツールはそれらを逐次処理することで，段階的に実行する．その結果，大規模なコードが生成できる．ユーザは，処理後のコードベースと Git のコミットログを確認することで，その変更内容を把握することができる．自律修正に失敗した場合は，テストツールが出力したエラーログもユーザに提供される．

コード生成ツールは，以下の四つのエージェントから構成されている．

1）依存解析エージェント

新規実装するコードと既存の実装の間の依存関係を設計書から解析する．具体的に利用するクラスが特定できる場合は，そのクラスの実装を抽出する．

2）類似コード検出エージェント

設計書の内容と類似した実装を持つコードサンプルを収集する．その理由は，他の実装を参考にすることでコード生成の精度を向上させるためである．現在はプロジェクト内のコードをサンプルとして利用しており，将来的には外部から手本となるコードサンプルを取得する機能も追加する予定である．

3）コード生成エージェント

収集されたコード情報を基に，新規にコードを生成する．プロダクションコードとテストコードの両方を生成し，それぞれが互いに影響を及ぼさないよう個別に処理する．これは，不正確なプロダクションコードがテストコードの生成に悪影響を与え，設計書やテストケースの要件に適合しないテストコードが生成されるリスクを避けるためである．個別に生成し，それぞれの結果を統合することで，コード生成の正確性を高めている．また，コード書き込みの後，Git でコミットを行い，更新履歴を記録する．

4）コード修正エージェント（4.4 節にて既述）

生成されたコードに対してテストを実行し，コンパイルエラーやアサーションエラーを検出し，コードを修正する．修正毎に Git でコミットを行い，更新履歴を記録する．

LLM の性質上，コード修正エージェントの自律修正能力には限界があり，常に正しいコードを生成するとは限らない．その際は，人間がコードを修正することになる．プロトタイプでは，自律修正の回数に上限を設け，修正回数が上限に達しても修正できない場合は，失敗とみなすようにした．人間によるコード修正の際には，コード生成ツールは Git のコミットログ，テストツールが出力したエラーログを提供し，修正を支援する．

6. 動作検証と効果の考察

本章では，プロトタイプの性能確認を目的として行った検証の結果を示し，考察する．検証の題材として，AlesInfiny Maia OSS Edition[*8] のサンプルコードのビジネスロジックの再現を試みた．この検証では，新規開発を想定している．サンプルコードは複数の Gradle Groovy DSL[*9*10] プロジェクトで構成されたマルチプロジェクト形式の Web アプリケーション（実装言語は Java）である．その中で，ビジネスロジックを担う「アプリケーションコア」プロジェクトを動作検証のターゲットとした．

検証に先立ち，プロジェクト内のパブリックメソッドから，Getter/Setter を除くメソッドに対して物理設計書とテストケースを作成した．入力となる物理設計書は本来人間が作成するものである．しかし，作成の補助のため，メソッドの正解の実装と物理設計書のサンプル 1 組を LLM に与えて生成したものを修正して用いた．また，テストケースは，物理設計書をテストケース生成ツールに入力し，生成された複数のケースの中から適切なものを選別して用いた．

検証環境のオペレーティングシステムは Windows 10 で，LLM には Azure OpenAI Service[*11] でリソースを作成した GPT4o-mini モデルを使用した．プロトタイプは自律修正時に Gradle でコンパイルコマンドまたはテストコマンドを反復実行する．この検証では，各コマンドの反復回数上限を 5 回とし，上限回数内での自律修正ができない場合には失敗とみなし自律修正を終了する設定とした．

6. 1 プロトタイプの検証結果

本プロトタイプの検証結果として，完全自動生成を達成したメソッドが存在する一方で，一部のメソッドでは完全自動生成が困難であることが確認された．ただし，完全自動生成を達成できなかったメソッドにおいても，コンパイルエラーやアサーションエラーを自律的に修正する機能は有効に働いた．完全自動生成が困難であったメソッドに関しては，プロダクションコードは概ね生成できたものの，テストコードの生成過程に問題があり，結果としてプロダクションコードの正確性が十分に保証されていなかった．また，「大きく外した」プロダクションコードが生成されなかった点や，自律修正機能が一定の効果を発揮している点から，静的なワークフローを採用したアーキテクチャが有効であることが示唆された．

一方で，いくつかのエージェントには課題が見られた．類似コード検出エージェントにおいては，参照すべきコードの検出が不十分なケースが確認された．これは，本検証が新規開発を対象としていたため，コードベース内に類似コードが多く存在しなかったことが原因と考えられる．5 章でも述べたように，外部から模範となるコードを取り込む機能の追加により，この問題への対処を検討している．

さらに，コード生成エージェントおよびコード修正エージェントでは，テストコード生成において，テストデータの設定およびプロダクションコードからの戻り値の検証方法に課題が

残っていた．たとえば，前提条件の異なる二つのテストケース群が存在する場合，一方のテストケース群にのみ有効な前処理を全体テストに適用してしまい，一部のテストのみ通過し他のテストが通らない状況が発生した．このようなケースでは，本来テストコード側を修正すべきであるにもかかわらず，LLM が問題をプロダクションコード側と誤認し，プロダクションコードを改変（改悪）する事象が確認された．これは，テスト失敗時にプロダクションコードとテストコードの問題を切り分ける仕組みが，本プロトタイプ内に整備されていないことが原因である．また，テストコードにおいて，プロダクションコードが返却するインスタンスのすべてのフィールドを検証していないため，バグを含むコードがユニットテストを通過してしまう事例も複数確認された．現行のプロトタイプでは，ユーザがテストケースにおいて詳細な検証内容を明示することで対処できる．しかし，ブラックボックス的なテストケース記述が求められる状況下では，特定インスタンスに対する精密な検証指示が困難となる．人の作業負担を増やさずにテストコード生成の信頼性を上げるため，エージェントへの理想的なテストサンプルの提供，エージェント内部でのテストコードの妥当性の自己レビューなど，複数のテスト精度向上策を検討している．

6.2　プロトタイプがもたらす効果

次に，本プロトタイプが実開発現場にもたらす効果について考察する．まず，本プロトタイプが生成したコードについて，時間あたりの有効行数[*12] を実際の製作工程での成果と比較した．コード生成ツールの総実行時間は約 35 分，生成された有効行数は 87 行であり，これは約 149.1SLOC/人時に相当する．一方，新規開発における標準的な生産性指標として，IPA の調査結果によれば，開発 5 工程の総工数（中央値：7395 人時，平均値：24458 人時）の内，製作工程の割合の中央値は約 33.0%，平均値は約 31.6%である．SLOC の中央値は 31.8KSLOC，平均値は 152.2KSLOC である[2]．これらの数値から生産性を推定すると，中央値ベースで約 13.0SLOC/人時，平均値ベースで約 19.7SLOC/人時となり，本プロトタイプは単純なコード生成速度の観点で，人間の開発者を 7 倍以上上回ることが示された．

また，実行時間および費用面についても考察する．物理設計書 1 件あたりの実行時間は最大 8 分程度であり，これも自律修正の反復回数が多い場合に限られる．実行時間が短いことには，コマンド実行回数上限の少なさや，高速なモデルを用いたことが寄与している．また，1 件実行あたり，一つのパブリックメソッドとそれに対応する複数のユニットテストが実装されるため，人手作業と比較して非常に高効率である．加えて，Azure OpenAI Service の API 費用も 1 件あたり約 0.6 ドル[*13] 程度に抑えられ，人件費と比較して経済的である．この結果から，本プロトタイプは人間と比較して生産性，費用対効果の両面で有利であることが示唆された．

最後に，将来的な開発プロセスの変容について考察する．LLM を活用したコード生成ツールは，人間のコーディング速度を大幅に凌駕する性能を示した一方で，エラーの自律修正機能には課題が残されている．そのため，生成されたコードに対して人手による修正を行う場面は依然として存在する．しかし，LLM を活用したツールの能力の高さから，今後の製作工程は LLM を中心としたものとなることが予測される．また，LLM が生成するコード量の増大から，人間がすべてのコードを精読することは困難と予測される．このような状況下では，自動テストが一層重要となる．すなわち，人間は自動テストの結果からコードベース内の問題箇所を特定し，修正を行うとともに，自動テストで十分にカバーできていない部分を補完的に確認する

役割を担うと考えられる．まとめると，人間の関与は従来のコーディング作業から，テスト駆動のコードレビューや修正作業へとシフトし，テスト結果を指針としてコードベース全体の品質保証を行うプロセスへの移行が進むものと推察する．本検証を通じ，現時点ではコードの完全自動生成が困難であることが明らかとなった．一方で，本アプローチの有用性を一定程度確認するとともに，将来の開発プロセスを展望することができた．検証で顕在化した課題を段階的に解消することにより，完全自動生成の実現に一歩近づく可能性があると考える．

今回取り扱ったサンプルコードは，比較的単純なロジックで構成されており，そのような条件下では，数年以内に完全自動生成を実現することは十分に可能な印象を得た．一方，LLM に品質保証の全責任を委ねることは依然として困難であり，人間によるコードベース全体に対する品質担保作業の継続は不可欠であると判断する．

7. お わ り に

本稿では，LLM を活用したコード自動生成ツールの可能性と課題について述べ，開発中のプロトタイプの概要と動作検証の結果を紹介した．コード自動生成は，特に大規模な開発現場において作業効率の向上に貢献するものと期待されている一方，LLM の不確実性や生成コードの正確性保証の難しさといった課題も明らかとなった．

プロトタイプの開発では，複数のエージェントが連携して情報収集，コード生成，エラー修正を行うことで，正確なコードの効率的な生成を目指した．しかし，動作検証の結果からも示されるように，完全な自律生成は実現に至っておらず，特にテストコードの生成に課題が残っている．今後はテストコード生成の精度向上に取り組み，より安定した自律的なエラー修正機能の実現を目指す予定である．

LLM を用いたコード生成技術は，今後も進化し，開発現場での実用性を高めていくと思われる．簡単な条件下であれば，完全自動化も夢ではない．しかしながら，実業務を想定すると完全な自動化にはまだ遠く，人間の介入を適切に組み合わせた仕組みづくりが重要である．その際アプリケーションには，LLM の行った膨大な作業の中から人間が適切な情報を確認できるように，介入を補助するような機能が求められる．筆者らは開発者に作業ログを提供する機能を実装した．プロトタイプがより大規模なコードを生成するようになれば，そのときはコードベース全体にフォーカスしたガイドが求められる可能性もある．人間が LLM の行った作業を容易に把握できるように，適切な可視化にも努めていきたい．

本稿が LLM を用いたアプリケーションの正確性を担保する仕組みづくりや，LLM を活用したコード生成のアプローチについて知るための一助となれば幸いである．

最後に，本稿執筆にあたりご協力いただいた関係者各位に深く御礼申し上げる．

＊ 1　GitHub Copilot：GitHub 社が提供する開発者向け AI ツール．https://github.com/features/copilot
＊ 2　Cursor：Anysphere 社が提供する AI 搭載型コードエディタ．https://www.cursor.com/
＊ 3　Devin：Cognition 社が提供する AI ツール．https://devin.ai/
＊ 4　OpenHands：All Hands AI 社のオープンソースソフトウエア．以前は OpenDevin という名称で知られる．https://github.com/All-Hands-AI/OpenHands
＊ 5　CodeAGI：SOPPRA DX 社が提供する AI ツール．https://www.codeagi.ai/
＊ 6　ANTLR：構造化文章の処理のためのパーサジェネレータ．https://www.antlr.org/
＊ 7　LangChain：LLM アプリケーションのためのフレームワーク．https://www.langchain.com/

＊ 8　AlesInfiny Maia OSS Edition：BIPROGY が設計した Java アプリケーションの一般的なアーキテクチャや方針設計のためのドキュメントとサンプルコードをここで提供している．https://maia.alesinfiny.org/
＊ 9　Gradle：Java 向けビルドツール．https://gradle.org/
＊10　Gradle Groovy DSL：Gradle で使用される Groovy ベースのドメイン固有言語．プロジェクトの依存関係やビルドタスクを簡潔に定義できる．https://docs.gradle.org/current/dsl/index.html
＊11　Azure OpenAI Service：Azure 上で生成 AI サーバを構築できるサービス．https://azure.microsoft.com/ja-jp/products/ai-services/openai-service
＊12　プロトタイプが生成したプロダクションコードの内，人間がコード修正を行った後も残ったコードの行数を「有効行数」として数えた．括弧やコメントのみの行や空行は除いている．また，テストコードも除いている．
＊13　2024 年 11 月 8 日時点での価格．（Azure OpenAI Service の価格 https://azure.microsoft.com/ja-jp/pricing/details/cognitive-services/openai-service/#pricing）

参考文献

［1］ 福地修一，吉野良成，「UNIX 環境における業務アプリケーション構築」，ユニシス技報，BIPROGY，通巻 38 号，1993 年 8 月，pp.84-101
［2］ ソフトウェア開発分析データ集 2022，独立行政法人情報処理推進機構，2022 年 9 月，https://www.ipa.go.jp/digital/software-survey/metrics/hjuojm000000c6it-att/000102171.pdf
［3］ Sander Schulhoff and Michael Ilie and Nishant Balepur and Konstantine Kahadze and Amanda Liu and Chenglei Si and Yinheng Li and Aayush Gupta and HyoJung Han and Sevien Schulhoff and Pranav Sandeep Dulepet and Saurav Vidyadhara and Dayeon Ki and Sweta Agrawal and Chau Minh Pham and Gerson C. Kroiz and Feileen Li and Hudson Tao and Ashay Srivastava and Hevander Da Costa and Saloni Gupta and Megan L. Rogers and Inna Goncearenco and Giuseppe Sarli and Igor Galynker and Denis Peskoff and Marine Carpuat and Jules White and Shyamal Anadkat and Alexander Miserlis Hoyle and Philip Resnik，The Prompt Report: A Systematic Survey of Prompting Techniques，arXiv:2406.06608 [cs.CL] 06，arXiv，2024 年 6 月，p.23，https://doi.org/10.48550/arXiv.2406.06608
［4］ Chen Qian and Wei Liu and Hongzhang Liu and Nuo Chen and Yufan Dang and Jiahao Li and Cheng Yang and Weize Chen and Yusheng Su and Xin Cong and Juyuan Xu and Dahai Li and Zhiyuan Liu and Maosong Sun，ChatDev: Communicative Agents for Software Development，arXiv:2307.07924 [cs.SE]，arXiv，2023 年 7 月，pp.1-13，https://doi.org/10.48550/arXiv.2307.07924
［5］ Wenlong Huang and Pieter Abbeel and Deepak Pathak and Igor Mordatch，Language Models as Zero-Shot Planners: Extracting Actionable Knowledge for Embodied Agents，arXiv:2201.07207 [cs.LG]，arXiv，2022 年 1 月，pp.1-33，https://doi.org/10.48550/arXiv.2201.07207

※　上記注釈と参考文献に含まれる URL のリンク先は，2024 年 11 月 25 日時点での存在を確認．

執筆者紹介

町　田　　凌（Ryo Machida）
2023 年 BIPROGY（株）入社．開発推進本部（旧プロダクトサービス第一本部）にて marbleMe の開発に取り組む．その後，12 月より，LLM によるソースコード生成に関する調査を担当．2024 年 4 月より，当社の技術戦略活動にてソースコード生成ツールの開発リーダーを務める．