生成 AI を利用したプログラミング検証結果について

日販テクシード株式会社

Consulting & Integration 統括本部

作成日：2024 年 1 月 15 日

【目次】

【図表目次】

1　はじめに

ChatGPT を筆頭に、大規模言語モデル(LLM)をベースとした多くの生成 AI が誕生し、全世界でビジネスに繋がる活用方法の検証が行われている。

生成 AI は様々な分野での活用が期待されているが、発展途上の技術であるという事もありまだ精度の面などにおいて多くの課題がある。しかしながら、IT 分野においては生成 AI と相性がよく、特に、自然言語特有の曖昧さの少ないプログラミング領域においては、法人・個人問わず、生産性を向上させることが出来る旨の報告がインターネット上に多く公開されている。

弊社においても、現時点の生成 AI のポテンシャルを確認することで、どういったことが出来るのかを理解し、システム開発におけるプログラミング領域において、生産性や品質の向上に繋がるかを判断するために、以下のような観点で検証ならびに評価を実施した。

- 生成 AI の利用方法　　※意図する結果を得るためのプロンプト
- 生成 AI が生成するソースコードの精度／品質
- 生成 AI が生成するソースコードの癖／留意事項
- 弊社におけるシステム開発現場での効果的な活用方法

2　検証結果の概要(要約)

生成 AI を用いたプログラミングの検証結果から、以下のことが明らかになった。

- 生成 AI は、コーディング経験の少ないエンジニアでも、プロンプトを作ることが出来れば要件に沿ったソースコードの開発が可能である。
- 処理効率に関しては、想定以上にバラツキがあり、処理結果に影響のある変更となる場合がある。
- 生成 AI を活用した開発では、従来のシステム開発とは異なる開発プロセスや、新しいスキルが求められる。

■　具体的な考察

- 処理効率の向上には、プロンプトで具体的なアルゴリズムや方法を指示する必要がある。
- 生成 AI の活用により、コーディング経験の少ないエンジニアでも開発が可能となるが、コンピューターサイエンスの知識がないと潜在バグの作り込みや技術負債の増加といったリスクにつながる。
- 生成 AI をシームレスに活用できるよう、モダンな開発プロセスにブラッシュアップしていく必要がある。
- 生成 AI の出力を正しく評価・修正するためには、プログラム構造だけでなく、プログラムやシステム全体の振る舞いの正しさを中心にテストを行う必要がある。
- 生成 AI を活用した開発では、テスト自動化を基本としたシステム開発プロセスの構築が重要である。

■　今後の課題

- 生成 AI の特徴を踏まえ、意図した結果を得るためのプロンプトスキルの向上
- 生成 AI の出力を正しく評価・修正するための知識とスキルの向上
- システム全体の品質を保証するためのテスト環境の構築

■　結論

生成 AI は、システム開発の効率化と品質向上に大きく貢献する可能性を秘めている。しかし、生成 AI を活用した開発では、従来のシステム開発とは異なる開発プロセスや、新しいスキルが求められるため、自社の強みを活かした開発プロセスを迅速に構築していく必要がある。

3　検証方法詳細

3.1　検証方法

効果的な検証を行うために、参加するエンジニアのスキル、利用するプログラミング言語、生成されるソースコードに対する仮説に基づいて、検証課題の準備を行った。本章では、検証を行う条件、検証に用いた課題の内容などについて記載する。

3.1.1　検証への参加方法と属性について

今回の検証では、「生成 AI を用いて要件を実現するコードを開発するチャレンジ企画」として、弊社エンジニアに対して参加希望者を募り、普段からプログラミングを実施している 20 代のエンジニアから新入社員(2023 年 4 月入社)までの 16 名にて検証を実施。なお、弊社では Java 言語によるシステム開発経験者が多いことから、生成するソースコードは Java 言語に限定としている。

3.1.2　生成 AI の使い方について

本検証では、利用する生成 AI の種類によってソースコードに違いがあるかについても評価対象であることから、利用する生成 AI に対する制約は設けていない。また、生成 AI の使い方で結果がどのように異なるかについても評価したいと考えていたため、チャレンジ方法として、「生成 AI のみで課題を解決する」「生成 AI の結果を利用しながら解決する」のどちらかを選択できるようにした。なお、一部のプログラミングスキルの高いエンジニアについては、「生成 AI を利用せずに解決する」を依頼した。

3.2　検証課題

3.2.1　検証課題について

参加したエンジニアは全員プログラミング経験者であるが、スキルレベルについては差があるため、プログラミング経験があれば解くことが可能な難易度の課題を設定。ただし、アルゴリズムによっては、処理効率やパフォーマンスを大きく改善することが可能な課題とすることで、結果にバラツキが出るようにすると同時に、生成 AI が非機能に対する考慮をどこまで行うかについても評価が行えるようにした。その他に、実際の業務では処理対象のデータの入力方法や仕様が決められていると同時に、処理を行った結果の出力方法も決められているケースが多くあるため、検証課題では本観点も含めて課題の作成を行った。

3.2.2　検証課題内容

実際に出題した検証用の課題は、以下の通り。

■要件

テキスト文書に対して、指定した複数の英単語で検索を行い、そのヒット件数とヒットした全英文を取得する

■テキスト文書の仕様

・文書は 1 つ以上の英文で構成され、各英文は改行文字によって区切られている

・英文は 1 つ以上の英単語で構成され、各英単語は空白文字によって区切られている
・文書は小さなサイズから、最大 500MByte 程度

■検索要件

・指定される検索英単語は 1～500 件
・ヒット件数が 0 件の場合、ヒットした英文は不要
・検索英単語と文書内の英単語は完全一致であること
　例：「host」で検索した場合、「hosts」はヒットしない
・1 つの英文に検索英単語が 2 つ以上存在した場合、ヒット件数は 1 件とする

■開発スコープ

KeywordSearcher クラス内に定義する、次のメソッドに対して実装する

```
public List<ResultRecord> search(File textFile, List<String> keywords) throws IOException
```

・File textFile
　検索を行う対象のテキスト文書の File クラス
・List<String> keywrods
　検索英単語の List 型で、本 List 型に含まれる全英単語に対して検索を行う
・List<ResultRecord>
　検索を行った結果を格納する ResultRecord クラスの List 型。ResultRecord クラスは検索英単語の検索結果を 1 件保存するための DTO で、検索を行った英単語(String 型)とヒットした英文(List<String>型)向けの Setter/Getter メソッドを持つ

■開発条件

・Java のバージョンは原則として JDK 17.0.9
・外部ライブラリの利用は不可
・開発を行う際の IDE(統合開発環境)は任意のものを利用可
・成果物は javac コマンドにてコンパイルを行い、java コマンドにてアプリケーションを実行
・main メソッドなどの実行に必要な最低限の機能を実装したソースコードに対して検索処理を追記
・他の参加者とのコミュニケーションは不可

3.3　課題の確認方法

3.3.1　納品物の受領条件

課題に対して正しく評価を行う必要があるため、次のテストにパスした納品物のみを受領条件とした。

・受領したソースコードがコンパイルできること
・提供済みのサンプルテキスト文書と検索英単語で正しく検索が行えること
・動作確認用のテキスト文書と検索英単語で検索を行い、予め準備していた正しい検索結果と比較して、差異が無いこと

テストをパスできないソースコードもあったが、正しい検索結果との差異を伝え、修正してもらうことで、全エンジニアの納品物を受領。

3.3.2　納品物の計測

今回の検証課題はアルゴリズムによって処理速度やコンピューターリソースの使用量に大きな差が出てくるため、生成されたソースコードの評価を行う上での補足情報として活用することを目的として計測を実施。

計測方法は、Amazon Web Services(以後、AWS)上に構築したサーバ上で、8パターンのテキスト文書に対して検索を実施し、処理時間ならびに使用リソース量の計測を実施。使用リソース量はアプリケーションを実行している間、1秒間隔でCPU使用率とメモリ使用量を取得した合計値から算出した利用料金とした。

表 1．計測に利用した文書のボリューム

	文書の行数	文書の単語数	文書のByte数	検索単語数	行数×検索単語数
パターン1	10,000	75,215	376,299	10	100,000
パターン2	100,000	1,753,802	9,647,056	30	3,000,000
パターン3	500,000	8,757,759	43,789,941	20	10,000,000
パターン4	1,000,000	17,504,696	105,028,176	100	100,000,000
パターン5	1,000,000	17,504,696	105,028,176	500	500,000,000
パターン6	2,000,000	35,012,812	210,076,872	100	200,000,000
パターン7	2,000,000	35,012,812	210,076,872	200	400,000,000
パターン8	5,000,000	100,007,562	500,038,476	500	2,500,000,000
合計	11,610,000	215,629,354	1,184,061,868	1,460	3,713,100,000

4　評価方法について

4.1　客観的評価指標

「生成 AI のみで課題を解決する」「生成 AI の結果を利用しながら解決する」「生成 AI を利用せずに解決する」に対して、テスト合格率、パフォーマンス、品質(可読性)の観点で評価を実施。

表 2．評価指標

指標	観点	備考
テスト合格率	実行結果が機能要件通りとなっていること	
パフォーマンス	・処理時間 ・CPU／メモリ平均使用率 ・CPU／メモリ利用コスト	AWS EC2 - t3.large (2vCPU／メモリ 8GiB)を用いて測定 ※計測方法は「3.3.2 納品物の計測」参照
品質（可読性）	コードメトリクスの計測	測定には、 Checkstyle を使用

ソースコードの品質を客観的に評価するために、ソースコードの複雑性を表す指標の 1 つである Cyclomatic Complexity（循環的複雑度）を用いる。複雑度の目安は以下の通り。

表 3．循環的複雑度の目安

循環的複雑度	複雑さの状態	バグ混入確率
10 以下	非常に良い構造	25%
30 以上	構造的なリスクあり	40%
50 以上	テスト不可能	70%
75 以上	いかなる変更も誤修正を生む	98%

また、生成 AI を用いたソースコードのレビュー活用の観点で、以下のような評価項目で生成 AI にてソースコードの評価を実施し、各評価項目を 100 点満点で採点を行った。

- 検索方法、アルゴリズム、高速化のための工夫
- ソースコードの可読性
- エラーハンドリング
- パフォーマンス

4.2　ユーザー主観の評価指標

利用した大規模言語モデルによる違い、プロンプトの分析を目的として、次の項目について選択回答＋回答理由の形式でアンケートを実施した。

- 利用した大規模言語モデル
- 大規模言語モデルの活用による、"生産性・品質が向上する" といった実感の有無
- 生成されるコードの特徴
- 生成 AI を活用する際の注意点
- プロンプトを考える際に意識したこと

5　検証結果

5.1　客観的評価

各チャレンジ方法におけるテスト合格率、パフォーマンス、品質(可読性)の結果は以下の通り。

表 4．定量評価

		生成 AI のみ	生成 AI を利用	生成 AI を未利用
テスト合格率		100 %	100 %	100 %
処理時間	平均	0 時間 35 分 24 秒	1 時間 11 分 29 秒	0 時間 14 分 16 秒
	最小	0 時間 01 分 55 秒	0 時間 48 分 37 秒	0 時間 04 分 34 秒
	最大	1 時間 50 分 08 秒	1 時間 35 分 25 秒	0 時間 23 分 59 秒
リソース利用料(CPU／メモリ)	平均	6.436 円	11.782 円	2.984 円
	最小	0.355 円	8.111 円	0.751 円
	最大	18.232 円	16.079 円	5.217 円
Cyclomatic Complexity	平均	6.750	5.667	5.500
	最小	5.000	4.000	5.000
	最大	9.000	9.000	6.000
生成 AI によるコード評価(4 評価合計)	平均	357.5	375.0	355.0
	最小	300.0	370.0	340.0
	最大	380.0	380.0	370.0

5.2　ユーザー主観の評価指標

5.2.1　利用生成 AI

参加者が利用した生成 AI は以下の通り。

表 5．利用生成 AI

利用生成 AI	利用率
Chat GPT	69 %
Phind	15 %
Chat GPT / Phind	8 %
MS Bing	8 %

5.2.2　生成 AI を利用した開発について

今回の検証で生成 AI を利用した結果に対するアンケート結果は以下の通り。

現場で生成 AI を利用する場合における、生産性と品質の観点で自由回答してもらった結果、ほとんどの回答者から生産性ならびに品質向上に生成 AI が寄与するとの回答があった。特に生産性向上については、処理内容から生成 AI でソースコードを生成する方がエンジニアによるコーディングよりも大幅に時間を短縮できる点が多く挙げられた。品質については、入力ミスや文法エラーといったヒューマ

ンエラーを軽減できる反面、処理の細かい修正が必要になったり、プロジェクトの規定に沿った実装(エラーハンドリング、ログ出力など)への手直しが必要になったりするといったコメントもあった。なお、品質課題についてはコーディング時間短縮で確保できた時間を品質向上に向けた活動に利用できるため、結果的には品質の向上に繋がるとの見解で、トータルで見ると生成 AI の活用により、生産性ならびに品質が向上するとの意見が多かった。

生成 AI を用いた開発を行う場合に注意すべきポイントについては、生成されるソースコードが要件を満たしていない可能性があったり、非機能要件の観点で考慮が足りていなかったりするため、ソースコードのレビューやテストについては今まで以上に重点的に実施する必要があるとの意見があった。その他に、プロンプトが曖昧の場合、生成結果にもブレが生じることから、適切なソースコードを生成させるためのプロンプトに関するノウハウを蓄積し、改善を行い、検証し、共有していくことでブラッシュアップを進めることが必要であるとの意見もあった。

6　検証結果からの考察

6.1　実装に対する考察

今回の検証では、処理効率の観点で 50 倍以上の差があったため、この大きな差について、実装アルゴリズムとプロンプトの観点から考察を行う。

表 6．処理効率結果

	最短/最小	最長/最大	ギャップ
処理時間	1 分 55 秒	1 時間 50 分 8 秒	57.98 倍
リソース利用料	0.355 円	18.232 円	51.35 倍

今回の課題は単純な文字列照合処理であるため、「繰り返し処理回数の削減」と「高速な文字列照合」を両立させる必要があり、ハッシュを上手く利用できるかが処理効率の大きなポイントとなる。
最も処理時間の短い実装は、文書から取得した 1 行を String.split で単語に分割し、検索英単語が 1 行の中に含まれているかを HashSet.contains で検索することでハッシュ値から直接検索英単語の有無を判断できるため、計算量は「文書の行数×検索英単語数×1」となる。
一方、次に処理時間の短い実装(3 分 8 秒)では、文書から取得した 1 行を String.contains でフィルタリング(部分一致)し、フィルタリングされた各行に対して検索英単語と完全一致している行のみを検索するアルゴリズムとなっており、String.contains は先頭から検索英単語と同じ文字列が含まれていないかをチェックするため、計算量は「文書の行数×検索英単語数×検索英単語にヒットするまでの平均文字数」となる。
なお、最も処理時間がかかっている実装(1 時間 50 分 8 秒)では、文書から取得した 1 行を String.matches で検索英単語と完全一致しているかをチェックしているため、計算量は「文書の行数×検索英単語数×1」と最も処理時間が短い実装と同様だが、「文書の行数×検索英単語数」回、処理に時間を要する正規表現をコンパイルとマッチングを行っているため、多くの時間を要する結果となっている。

6.2　プロンプトに対する考察

生成 AI へ指示を行うプロンプトによって生成されるソースコードの違いについて考察を行う。
今回、唯一ハッシュ探索法を用いたソースコードは、処理概要・条件・main クラス・メソッド定義・サンプルデータによる実行例と結果を 1 回の指示で提供して生成されたコードであった。なお、同等の記述レベルのプロンプトもあったが、ハッシュ探索法を用いたソースコードは生成されなかった。
また、全く同じプロンプトで 1 ヵ月以上経過後にソースコードを生成したが、ハッシュ探索法を用いたソースコードは生成されなかった。これは、生成 AI の言語モデルが日々更新されており、同じ条件であっても結果が異なるためであると考えられる。

その他のプロンプトについては、Markdown 形式で構造化されたものや、数行程度の簡単な要件の指示のみであっても、ある程度要件を満たすソースコードが生成され、想定と異なる挙動については追加で修正指示を行うことで対応していた。生成されたソースコードに対して追加で修正指示を行ったも

ので、最も多かったのが完全一致検索となるようにする指示であった。また、複数回指示を行ってソースコードを生成する場合、最初に生成されたソースコードをベースとして修正を行うため、最初に処理効率の悪いソースコードが生成されると、引き続き処理効率が悪い状態となる特性が見受けられた。

6.3 生成 AI を利用した開発に対する考察

今回の検証結果から、生成 AI を利用して開発を行うケースについて考察を行う。
前述の検証結果から、処理効率に起因する処理時間とリソース利用料を除く、正確性、ソースコードの可読性についてはバラツキが少なく、内容に大きな問題は見られなかったことから、生成 AI を活用することで、意図する要件を実現するソースコードを生産性高く開発することが出来ると考える。また、ソースコードの生成を行うためのプロンプトに記述する要件についても、仕様の厳密さ、詳細な要件への落とし込みを行わなくても生成 AI は汎用的な要件を補完してソースコードを生成できるが、正確なハンドリングが必要な場合は、プロンプトに明確な機能要件を記述する必要がある。

処理効率に関しては、想定以上にバラツキがあり、さらに、プロンプトに扱うデータ量などを記述している場合でも、生成されるソースコードにはデータ量を考慮された形跡が見られなかった。そのため、処理効率を向上させるためには、プロンプトで具体的なアルゴリズムや方法を指示する必要がある。なお、要件を満たしたソースコードを生成後に、性能向上の改善指示をすると、生成 AI はリファクタリングを行い、検索単語単位でのマルチスレッド化による改善を行ったが、処理結果に影響のある変更であったため、追加でソースコードの修正指示を行う必要があった。改善対象のソースコードは線形探索法による検索であることから、マルチスレッドによる高速化が有効であると考え、本ソースコードについては、スケールアップした環境にて実行し、性能が改善されていることを確認した。

表 7．実行環境スペックの違いによる性能改善評価

実行環境	処理時間	平均 CPU 使用率 (Max:200%/800%)	平均メモリ使用量	リソース利用料
t3.large (2vCPU/8GiB)	1 時間 8 分 11 秒	195%	2,000 MB	15.950 円
t3.2xlarge (8vCPU/32GiB)	18 分 9 秒	782%	5,585 MB	19.292 円
増加率	26.59%	401.88%	279.27%	120.95%

生成 AI の活用により、コーディング経験の少ないエンジニアであっても、プロンプトを作ることが出来れば要件に沿ったソースコードの開発が可能であることが分かったが、実務で利用する際には、コンピューターサイエンス（情報工学）を理解していないと、潜在バグの作り込みや技術負債の増加といったリスクにつながることも分かった。コンピューターサイエンスは、プログラミング、アルゴリズム、データ構造、デザインパターン、アーキテクチャ、OS、コンピューター構造、ソフトウェアエンジニアリング（分析、設計、構築、テストなどのシステム開発プロセス）、セキュリティといった、システムエンジニアに求められる基礎理論やコア技術の分野のことで、生成 AI で生成されたソフトウェアが、システム特性と照らし合わせた場合に「適切」な実装になっているかは、コンピューターサ

イエンスを学んだエンジニアでなければ評価するのが難しい。例えば、今回生成されたソースコードについては、以下のポイントで評価を行うことが想定される。

表 8．生成されたソースコードに対する定性評価

	評価ポイント	評価/コメント ()内はコンピューターサイエンスにおける該当分野
1	検索アルゴリズム	線形探索法による検索、ハッシュ探索法による検索 (アルゴリズム、デザインパターン)
2	文書データをすべての変数に保持を行ってからの検索	以下の観点から、処理に必要な情報のみ変数に保持を行う方式に変更するべきである ・バッチ処理で文書データの再利用は行わない ・ディスクキャッシュによる高速化の効果が期待できる検索アルゴリズムを利用している ・最大 500MB のデータを扱うためメモリを多く消費する ・クラウド環境下ではリソースとコストのバランスが必要 (アルゴリズム、データ構造、OS、コンピューター構造)
3	正規表現を用いた検索	完全一致による検索に加え、繰り返し一致していることの比較を行うため、計算量の多い正規表現の利用は避ける (アルゴリズム)
4	検索単語単位によるマルチスレッド化	検索単語数は最大 500 個指定可能なことから最大 500 スレッド生成されるが、Executor を利用しているためスレッドプールによるスケジューリングで適切なリソース管理が行えている。また、排他制御などは不要な要件であるためシンプルな実行が行えるが、クラウド環境下ではスケールアップによる性能改善はコスト増につながる点を意識する必要がある。 (データ構造、デザインパターン、OS、コンピューター構造)

6.4 生成 AI を利用した開発プロセスについて

これまでの考察から、コーディング工程で生成 AI を活用する場合は、生成 AI をシームレスに活用できるよう、モダンな開発プロセスにブラッシュアップしていく必要があると考える。開発プロセスは企業やプロジェクトによって異なるため、ここでは、日本企業で広く採用されている IPA(独立行政法人 情報処理推進機構）のウォーターフォール型プロジェクトを想定して、設計、開発、テスト工程、保守開発についてブラッシュアップすべきポイントを考察する。

6.4.1 設計工程

プロジェクトにおける設計書は、自然言語による記述が多く、曖昧な表現となるため、細かい抜け漏れが発生しやすい課題が従来からあった。プログラマによる開発では、抜け漏れが発生していた場合

でも、コーディング中に処理を記述できないため、要件を確認したり、機能内容から適切な処理を記述したりすることで対応していたが、生成 AI は「プロンプトの内容から最も確率の高い実装を行う」ため、意図しない実装となるリスクが高くなる。そのため、機能要件を論理的にまとめた設計がこれまで以上に求められる。
その他に、設計書のテンプレートは Word や Excel といった Office ツールをベースとしているケースが多く、表記がヒューマンリーダブルであることから、プロンプト向けに変換して記述する必要がある。そのため、Markdown 表記、OpenAPI 表記、PlantUML 表記などを用いることで、ヒューマンリーダブルとマシンリーダブルの両立を図るアプローチを検討する必要がある。

6.4.2　開発工程

開発者は生成 AI を積極的に活用することで、生産性と品質を向上させる。生成 AI は、ソースコードの生成だけでなく、作成したソースコードの初回レビューや、開発のリアルタイムサポート(Copilot 機能)にも活用することができる。現状、生成 AI の出力は、開発者の意図した成果物ではない可能性があるため、それを意識した自己レビューを行う。
ソースコードレビューについては現状同様、システム特性、エラーハンドリング、プロジェクトで規定された開発規約、開発ガイドラインに沿った実装となっていることの確認を行う。

6.4.3　テスト工程

今回の検証において、生成 AI に対して追加指示を行った結果、処理結果が異なるケースが散見されたため、テストを重点的に実施することで、デグレードが起きていないことを担保し、アプリケーション品質を確保する必要がある。特に、生成 AI は常に進化しており、同じプロンプトでも生成されるソースコードが異なるため、プログラムやシステム全体の振る舞いの正しさを中心にテストを行うアプローチが適しており、従来のプログラム構造に焦点を当て改修内容に応じて影響範囲を特定してテストスコープを決定するアプローチとは大きく異なる。
なお、モダンな開発現場では、継続的なソフトウェアインテグレーション（CI）の考え方に基づき、高頻度でテストを行うため、テスト自動化を実現している。また、ソースコードなどの変更があった場合には、ソフトウェア全体のテストを行うアプローチを適用しているケースが多いことから、生成 AI を用いた開発との親和性が高い。従来の開発方法では、高頻度なテスト実施はコスト面で難しいため、生成 AI の活用と合わせてテスト自動化に取り組むことが重要である。
今後、システム変更が発生した場合、生成 AI を活用することで変更箇所とソースコードの改修の大部分を効率的に行えるようになることが想定されるため、テストが今まで以上に重要となる。そのため、テスト自動化を基本としたシステム開発プロセスの構築を検討する必要がある。なお、テスト自動化は初期開発コストにインパクトがあるため、開発規模、改修頻度などを加味した上で判断を行う必要がある。

6.4.4　保守開発

小規模な改修の場合、変更前のソースコードと変更したい内容をプロンプトで指示することで、開発工程のプロセスと同じ流れで対応を行うことができる。さらに生成 AI の活用が進み、RAG(Retrieval-Augmented Generation)を利用してプロジェクトの設計書やソースコードなどのデータに基づいて生成 AI が回答できるようになると、要件や設計情報の変更概要をプロンプトで指示することで、関連する

ソースコードを生成 AI にて修正を行うことが可能となる。しかし、ソースコードの修正を生成 AI に任せる場合、意図した変更が正しく実装されないリスクがあるため、テストによる品質確保がより重要となる。意図した実装と異なる場合、プロンプトにさらに具体的な情報を追加してソースコードを修正する、エンジニアが適切な実装に修正する、といった対応を行い、繰り返しテストを実施するサイクルを行う必要がある。そのため、テスト工程で構築したテスト自動化を活用することで、効率化と品質の確保を図ることが有効となる。

上記の考察の通り、生成 AI を活用したシステム開発では、

- 生成 AI とのシームレスな連携が行える開発プロセスの構築
- 生成 AI の特徴を踏まえ、意図した結果を得るためのプロンプトスキル
- 生成 AI の出力を正しく評価・修正するための知識とスキル
- システム全体の品質を保証するためのテスト環境の構築

といった、従来のシステム開発とは異なる開発プロセスや、新しいスキルが求められることになる。生成 AI を最大限活用するためにも、自社の強みを活かした開発プロセスを迅速に構築していく必要がある。

Appendix.

Appendix A. 定量評価結果詳細

No.	チャレンジ方法	処理時間 (時:分:秒.ミリ秒)	平均 CPU 使用率 (Max:200%)	平均メモリ使用量	リソース利用料	Cyclomatic Complexity	コード評価 (400 点満点)
1	生成 AI のみで課題を解決	0:01:54.953	137%	930 MB	¥0.355	6	370 点
2	生成 AI のみで課題を解決	0:03:08.432	148%	1,346 MB	¥0.646	8	350 点
3	生成 AI のみで課題を解決	0:03:42.290	123%	1,629 MB	¥0.661	8	350 点
4	生成 AI のみで課題を解決	0:04:07.177	122%	694 MB	¥0.675	5	300 点
5	生成 AI を利用しない	0:04:33.560	122%	837 MB	¥0.751	5	340 点
6	生成 AI のみで課題を解決	0:08:15.199	117%	699 MB	¥1.301	9	350 点
7	生成 AI を利用しない	0:23:58.680	113%	5,491 MB	¥5.217	6	370 点
8	生成 AI を活用して課題を解決	0:48:37.334	112%	1,909 MB	¥8.111	5	370 点
9	生成 AI を活用して課題を解決	0:48:46.882	112%	1,964 MB	¥8.176	5	370 点
10	生成 AI を活用して課題を解決	0:58:41.037	112%	793 MB	¥8.911	7	370 点
11	生成 AI のみで課題を解決	1:08:11.084	195%	2,000 MB	¥15.950	8	380 点
12	生成 AI のみで課題を解決	1:23:47.219	110%	1,606 MB	¥13.670	5	380 点
13	生成 AI を活用して課題を解決	1:27:43.806	112%	2,025 MB	¥14.794	4	380 点
14	生成 AI を活用して課題を解決	1:29:40.363	111%	1,710 MB	¥14.620	4	380 点
15	生成 AI を活用して課題を解決	1:35:25.430	112%	2,040 MB	¥16.079	9	380 点
16	生成 AI のみで課題を解決	1:50:07.890	111%	1,854 MB	¥18.232	5	380 点

※処理時間を昇順でソート

※実行環境：AWS EC2 - t3.large(2vCPU, 8GiB) / java 17.0.9(2023-10-17 LTS)

※処理時間は全 8 パターンの合計時間で文書英文行数と検索英単語数合計：文書英文行数 1,161 万行／検索英単語数 1,460 単語

内訳(文書英文行数(検索英単語件数))：1 万行(10) / 10 万行(30) / 50 万行(20) / 100 万行(100) / 100 万行(500) / 200 万行(100) / 200 万行(200) / 500 万行(500)

Appendix B. 参加者のアンケート結果（生成 AI の活用）

現場で生成 AI を活用する方法(コード生成、コードレビュー、Copilot から選択)とその理由について生産性と品質の観点で自由回答してもらった結果は以下の通り。

活用方法	理由
コード生成	1 から自分でコーディングするよりも、生成 AI でコーディングしたものをたたき台にして修正することで、生産性が向上すると思うため。しかしながら、品質が向上するかは利用エンジニアのリテラシーに依存するため、生産性と比較すると上がりにくいと思う。
コード生成	自分で 1 から作ったときと比較し、生成 AI だと 3 分の 1 ぐらいの時間で大まかなコードの生成ができた。そこから細かい処理の修正は必要だが、トータルの時間は生成 AI を活用したほうが速いと感じた。
コード生成	Python のコード生成で利用しているが、生成 AI を用いることで開発速度が 1.5～2 倍程度早くなっている感覚があるため。
コードレビュー	0 からソースコードを生成してくれるという部分で生産性は上がると思うが、コードを生成するには明確な指示が必要となる。しかしながら、すべてを生成 AI にコーディングさせるには、不足している指示の追加、生成されたコードの手直しが必要となるため、コードレビューとして活用するのが良いと思った。
コード生成	処理内容を言語化できれば、ソースコードを入力する手間が省けることと、打ち間違いなどの人為的なミスも減らせるため、活用できると思う。また、「高速化してください」「より効率化してください」といった漠然とした指示でも、多彩な方法が提示され、それをコードに反映できる点は便利であると感じた。
コード生成	要件を正確に伝えることが出来れば、希望するソースコードがすぐに生成されるため、生産性が向上していると感じたが、品質面では向上している時間はなかった。
コード生成	ネットで調べてもわからないと感じる慣例が数多くあるため、そういったときに利用するとすぐに解決できる時もあるため生産性が向上すると感じる。
コード生成	大まかな流れでも生成されるとコーディング時に取り掛かりやすいと感じた。また、行き詰った際にヒントとして気軽に利用できるのに加え、うまく利用する事で精度の高いものをすぐに生成してくれるため。
コード生成	自分でコーディングする手間が省ける、ある程度動作するコードを生成 AI から提案してもらうことで悩む時間が削減されるため、生産性が向上すると思う。自力でコーディングする時と比較して、文法エラー等で詰まる時間はほぼなかった。また、AI に修正指示を重ねていくことで品質も向上すると思う。

活用方法	理由
コード生成	業務での開発経験がほぼない自分でも、ある程度のものを短い時間で作成することが出来たため生産性は向上すると思う。しかし、品質については、基本的な考え方やアンチパターンなどをについて理解がないと品質の悪いコードになると感じた。開発速度が上がる分、品質チェックやテスト、コードレビューに時間をかけることができれば、結果的に生産性と品質を向上させることが出来ると考える。
コード生成	ある程度要件を絞って実装を指示することで、0 からコーディングを行う必要がなくなり、ある程度のスキルがあるエンジニアならコード生成の間に実装方法を考えることが出来る。生成されたコードを基に再実装する流れで生産性が向上する。さらに、要求を追加して様々な実装方法を見比べる、自分の考えと比較することで自分にはなかった着想を得ることで品質も向上すると考える。
コード生成	コーディングに関しては回答の精度が非常に高く、日本語対応もしているため、Q&A フォーラムから時間をかけて情報収集する手間が省けるため、生産性が向上すると思う。
Copilot	コーディング時に迷った場合に、Copilot によりサポートしてもらえ、生産性を向上できると考えるため。

Appendix C. 参加者のアンケート結果（生成 AI を活用する際の注意点）

生成 AI を活用して開発を行う場合において、留意する必要があると感じた点について自由回答してもらった結果は以下の通り。

注意点有無	注意すべきポイント
あり	生成 AI 利用時の注意事項としてアナウンスされているが、生成 AI の利用に慣れすぎると、無意識のうちにそのシステム特有の情報(社外秘の設計やソースコード)を入力してしまう点。
あり	生成 AI を使うと生産性は向上するが、保守も自社で行うケースが多いため、結局自分自身でソースコードを理解して修正する力も必要になり、生成 AI でのコード生成に慣れていないときは依存しすぎない方が良い点。
あり	ログ出力、エラーハンドリング、セキュリティ対策、性能、コードの保守性など。プロジェクトで規定されるコーディング規約やアーキテクチャの方針に沿った実装といった点を考慮しないと、本番のソースコードとして耐えられない可能性があるため。
あり	生成されたソースコードやレビューとして活用した場合に、鵜呑みにするのではなく検証が必要なため。
あり	指示の仕方が曖昧だと、実行のたびに結果にブレが生じ、適切なコードが出力されない場合があるため、プロンプトの入力方法についてはノウハウを蓄積・共有する必要がある点。
あり	細部で要件を満たしていない可能性があるので、要件を満たしているか正確にチェックしなくてはならないと感じたため。
あり	質問の仕方によっては生成されたコードが正確であるとは限らないので、あくまで実装の参考として活用する必要がある点。
なし	特に意識せず使用できたため。
あり	生成されたソースコードが機能/非機能要件を完全に満たす保証はないため、テストを重点的に実施する必要がると感じたため。
あり	処理効率や保守性の悪いコードが作成される可能性があるため、そのチェック体制が必要だと感じたため。
あり	最終的には、本当に問題が無いか確認する必要がある点。現状では生成 AI にすべてを任せられるほどのレベルではないため、レビュー&テストは必須である点であるが、実装の生産性が向上することで、品質チェックの注力に繋げられる。
あり	ChatGPT の場合、返答が正確でない場合がある点。
あり	プロンプトで指示をする方法について情報収集を行う必要がある点。

Appendix D. 参加者のアンケート結果（生成されたコードの特徴）

生成 AI が生成したソースコードに対して、気になる特徴があると感じた点について自由回答してもらった結果は以下の通り(「ある」と回答した参加者の回答のみ)。

感じた特徴
Phind は生成されるソースコードに誤りが少なく、ChatGPT と比較して安定しているイメージであった。ChatGPT はブレがあったが、よりよいコードを模索する上で様々な提案を得られたので、今回の検証では有効に活用出来た。
仕様を局所的に、また要約してプロンプトの入力とするより、そのまま入力したほうが生成されるコードの精度が高いと感じた。
感覚的ではあるが、詳細な指示になるほど、指示内容を忠実に実装しようとするソースコードが生成されているように感じた。
2 回目のプロンプト以降、再作成を指示しても、1 回目のプロンプトで生成したソースコードに依存した形で回答された。1 回目のプロンプトの回答時点で、アプローチから間違っている可能性などを考えると、何度かやり取りを重ねて一つのコードにたどり着くよりも、1 回目のプロンプトをより正確なものにして、1 回で良いソースコードを生成したほうが良いと感じた。
要件をきちんと指示しないと、力技で実装されてしまう事には注意が必要と感じた。しかし、コーディングに特化した生成 AI であれば、一般的にプログラマに求められる要素を踏まえたうえで要望に応えるため、大きな問題にはならないと思う。
無駄と思われる変数が宣言されることが多いと感じた。

Appendix E. 参加者のアンケート結果（他者の成果物を参照した感想）

全員の課題提出後、処理時間とリソース利用料の集計が行えたタイミングで、参加者全員の提出されたソースコードと生成 AI に指示したプロンプトを公開した。自身の指示と他者の指示を見比べることで情報共有ならびに多くの気付きを得られるようにし、その結果について自由回答してもらったコメントは以下の通り。

感じた特徴
他の人のプロンプトを見てみると「あなたは優秀なプログラマです」「あなたはプロの IT エンジニアです」といった役割を与えていることに気づいた。他にもどのようなコツがあるのか調べていきたい。
処理内容について具体的な指示が出来ている参加者ほど実行速度が速くなっているように思った。 また、生成されたソースコードに対して「処理速度を速くしてください」と言っても出来上がった処理をベースにするため、基が速くないと処理改善は難しいと感じた。 生成 AI のみでの実装の処理時間が最速であったため、自分も使いこなせるようにならないと、置いて行かれるという危機感を覚えた。
自分なりのプロンプトの定型文があったが、「このようなプロンプトでも同じ結果が得られるプログラムが生成される」ことは勉強になった。また、同じように生成 AI のみで作った人でも、ここまで性能に差が出ることは驚きだったため、案件で利用する場合は性能面をコードレビューしたほうが良いと感じた。
指示の違いで結果が大幅に変わる点は興味深かった。コードを書く部分では考える時間は省略されるかもしれないが、指示を考える時間を考慮すると、生成 AI を使っても使わなくても変わらないかもしれないとも思った。
自分よりも抽象的な指示で要件を満たしたコードが得られている成果物が多くあったため、いかに自分で考えずに AI を動かせるか、という点は今後も検討の余地がある課題だと感じた。
処理時間にここまで差が出るとは思っていなかった。プロンプトの指示の仕方次第でより優れたコード生成が出来るというのは大きな学びとなった。
自分は要件の内容をすべて記載し、あとは伝わるように加筆修正を行ったので、プロンプト量も多く、あまりまとまっておらず、わかり辛い構成になっていた。しかし、他の人のプロンプトは要件の内容をコンパクトにまとめて生成 AI に指示をし、望むような答えを引き出せていたので、プロンプトの作り方は改善しなければならないと感じた。
プロンプトの記載方法について、仕様書のように概要や条件を詳細に記載している参加者がいて参考になった。自分の場合、意図通りに動かすために複数回やり取りが発生したため、プロンプトはある程度指示のテンプレートを作成しておくと良いかもしれないと思った。
完成イメージをプロンプトに記載している方が、少ないやり取りで精度の高いソースコードが生成されているように感じた。 また、情報を整理してプロンプト指示をしないと、その分余計な処理が増えてしまうと思った。処理性能が良かった参加者のプロンプトは、情報がわかりやすく整理されていたと感じた。

感じた特徴
自分の場合、出題された要件や準備されたソースコードをプロンプトの指示にしただけであったが、他の「生成 AI のみで課題を解決する」参加者のプロンプトを見ると、生成 AI の役割を定義している手法が見られたので、勉強となった。 また、GPTs が出てきたことで、プログラマに特化した生成 AI を利用できるようになり、これから求められるスキルが何かを考えるきっかけとなった。 コーディング力があれば、生成 AI を活用することで数倍も早く＆品質も良く実装ができると感じた。コーディング力がなくても実装をある程度カバーできるため、レビュー&テストに開発リソースを割くことができる。また、ソースコードの“正しさ”を見極めることのできるスキルが重宝されると思われる。
他の参加者の成果物では非常に細かいテキストプロンプトを使って生成したものがあり、そのほうが簡潔なソースコードとなっていて、精度に違いがあるかもしれないと感じた。
プロンプトの指示内容が充実しているほど正確な回答が返ってくると感じた。Web 検索するように利用していたが、ChatGPT はより人間に近い存在と思って指示を出しても対応してくれると知ることができたため、利用方法を調べて改める必要があると感じた。

日販テクシード株式会社

〒103-0007　東京都中央区日本橋浜町３丁目３－２
トルナーレ日本橋浜町　オフィス棟 4F
URL　https://techceed-inc.com/