生成 AI を活用したコード生成の精度と効率化に関する検証と考察

Verification and discussion on accuracy and efficiency of code generation using generative AI

加古川 誠史[1] 杉本 雄太[1] 中山 真吾[1] 古本 拓也[1] 若林 佑[1] 追川 修一[1*]

Seiji Kakogawa[1] Yuta Sugimoto[1] Shingo Nakayama[1] Takuya Furumoto[1] Tasuku Wakabayashi[1] Shuichi Oikawa[1*]

[1]東京都立産業技術大学院大学 Advanced Institute of Industrial Technology
*Corresponding author: Shuichi Oikawa, oikawa-shuichi@aiit.ac.jp

Abstract This study investigates the effectiveness of generative AI in enhancing code generation accuracy and development efficiency of software programs. We examine the performance of leading large language models (LLMs), such as GPT-4o, Claude 3.5 Sonnet, and Gemini 1.5 Pro, on coding tasks using the HumanEval problem solving dataset provided by OpenAI. Our findings revealed that while individual LLMs achieved accuracy rates between 83.5 and 91.5%, combining these multiple models with ensemble techniques improved performance to 96.3%. We also explored prompt engineering techniques and its best practices to create high-quality prompts that optimize model's outputs and reduce hallucinations. Additionally, we evaluate the potential of the Sketch2code technology for front-end development by comparing the visual similarity of wireframes to generated HTML using metrics such as SSIM and PSNR. Our results indicated that Gemini 1.5 Pro performed the best outcome in this specific task, though all models showed comparable capabilities. Finally, we explored that the emergence of AI agents and their potential impact on software engineering field, referencing a new benchmark called SWE-bench (Software Engineering Benchmark). Agent-assisted models do not require human's instructions and significantly outperform traditional LLMs. We showed a case study using the Amazon Q Developer Agent to demonstrate efficiently creating a prototype of web application with the AI's capability.

Keywords large language models; code generation; prompt engineering; AI agents; sketch2code

1 はじめに

近年、大規模言語モデル（LLM）の発展により、ソフトウェア開発の分野においても人工知能（AI）の活用が急速に進んでいる。自然言語だけでなくスケッチからのコード生成も可能となってきており、開発者の生産性を飛躍的に向上させる技術が次々と登場している。本紀要では、これらの最新技術の性能評価と実用性の検証を行い、ソフトウェア開発における AI の有効活用について考察する。

本紀要の主な目的は、以下の 4 点となる。第一に、各種 LLM のコード生成性能を比較評価し、その特性や限界を明らかにすること。第二に、AI による高精度な出力を実現するための手法や技術を調査し、その有効性を検証すること。第三に、Sketch2code と呼ばれる、スケッチから HTML コードを生成する技術の現状を評価し、フロントエンド開発における有用性を検討すること。そして第四に、より実践的なソフトウェア開発の文脈で AI の性能を評価するための新しいベンチマークを探索することである。

まず、LLM によるコード生成の比較では、既存の評価指標を用いて、GPT-4o、Claude 3.5 Sonnet、Gemini 1.5 Pro の性能を分析する。特に、各モデルが不正解となった問題の傾向や、複数のモデルを組み合わせることによる正解率の向上の可能性についても調査を行う。これにより、現在の LLM が持つコード生成能力の到達点とその課題を明らかにする。

次に、AI による高精度な出力を実現するための手法・技術については、各生成 AI モデルに共通するベストプラクティスや、ハルシネーション（誤った情報生成）を減らすためのテクニックを検討する。さらに、これらの知見を活用して改良したプロンプトを用い、不正解だった問題に対する再試行を行い、その効果を検証する。

Sketch2code に関する調査では、最新の LLM を用いて、スケッチから生成された HTML コードの出力とワイヤーフレームの類似度を比較評価する。シンプルなプロンプトを使用した場合の各 LLM の特性を分析するとともに、前述の高精度の出力のための技術を適用したプロンプトでの再検証を行い、フロントエンド開発における LLM の有用性を総合的に判断する。

最後に、ソフトウェア開発に即した新しいベンチマークによる評価手法を用いた課題解決能力の評価、AI エージェントを活用したコード生成の可能性を探る。これにより、より実践的な開発環境での AI の性能を多角的に評価する。

本紀要は、急速に進化を続ける AI 技術のソフトウェア開発への応用の可能性を、最新の知見と実験データに基づいて多面的に検討するものである。コード生成、プロンプトエンジニアリング、AI エージェントなど、新しい技術トピックを取り上げ、それぞれの現状と課題、そして将来の展望を明らかにすることを目指す。本紀要の成果は、単なる技術評価にとどまらず、実際のソフトウェア開発プロセスにおける AI の効果的な活用方法や、開発者の役割の変化、さらには様々な仕事のあり方への影響など、より広範な議論につながる可能性を持っている。例えば、Amazon Q Developer や Visual Studio Code などの開発環境における AI 統合の動向も考慮に入れ、より包括的な視点からソフトウェア開発の未来像を描くことを試みる。

本紀要を通じて、ソフトウェア開発における AI の現在地を正確に把握し、その潜在的な可能性と課題を明らかにすることで、今後の AI 時代におけるソフトウェアエンジニアリングの方向性を示す一助となることを期待する。

2 LLM によるコード生成の比較

2024 年 9 月時点で一般に広く利用されている LLM のサービスとしては、OpenAI の ChatGPT、Anthropic の Claude、Google の Gemini が挙げられる。2024 年度 PBL の追川 PT においても、チームメンバーが上記の 3 つのサービスを併用して、ソフトウェア開発を実践している。この 3 つのサービスをどのように使い分ければ、効率的なコード生成が可能になるのか、それぞれのサービスのベースとなっている LLM モデルの特徴を比較する。最新の LLM モデルとしては、OpenAI が GPT-4o[1]、Anthropic が Claude 3.5 Sonnet[2]、Google が Gemin 1.5 Pro[3] をリリースしており、これらを比較の対象とする。

Received: 2024-10-30; Published online: 2025-02-28

コード生成性能の評価指標による比較

LLM によるコード生成の性能評価の指標としては、HumanEval というデータセットをベースにした評価が標準的になっている。HumanEval とは、OpenAI 社が 2021 年の論文[4]で発表したもので、コード生成の性能評価を行うため、Python のユニットテスト付きの 164 のプログラミング問題からなるデータセットであり、このデータセットには、回文、ソート、面積計算など基本的な問題が含まれている。各問題では、英文の関数シグネチャと docstring（ドキュメンテーション）が与えられ、LLM が生成したコードがユニットテストに合格するかどうかを判定する。

HumanEval のスコアは、Claude 3.5 Sonnet は 92.0%[2]、GPT-4o は 90.2%[5]、Gemini 1.5 Pro は 84.1%[3]と発表されていて、Claude 3.5 Sonnet が最も高い。Claude 3.5 Sonnet は、以前のモデルである Claude 3 Opus の 84.9%から大幅にスコアがアップしたと報告している[2]。

HumanEval のリーダーボードを掲載しているサイト[6]によれば、図 1 で示すように 2021 年に OpenAI が HumanEval の評価指標を提案した際のモデル Codex のスコア 8.2%から、この 3 年間で大きく性能が向上していることがわかる。Codex は、Github で公開されたプログラミングコード数十億行を訓練データとして、GPT-3 をチューニングして開発されたコード生成に特化したモデルで、その技術がコーディング支援サービス GitHub Copilot に活用されている[4]。ただし、コード生成の標準的な評価指標となっている HumanEval のデータセットについては、web 上に漏洩している恐れがあるため、データコンタミネーションによって、適切な評価ができなくなっているという懸念も指摘されている[3]。

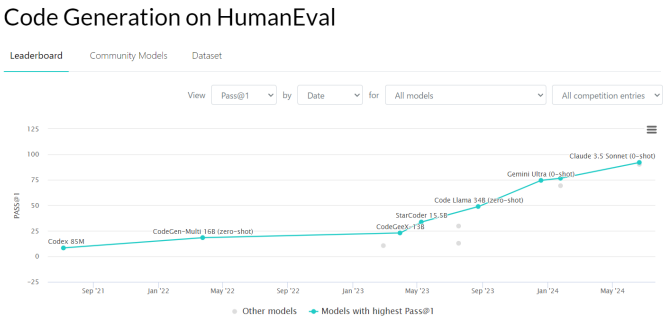


図 1　HumanEval のリーダーボード（2024 年 10 月 8 日現在）

HumanEval において各 LLM が不正解だった問題の比較

HumanEval のプログラミング問題（164 問）のうち、各 LLM がどんな問題を解けなかったのかを確認する。HumanEval のデータセットをダウンロードして、手元の PC で確かめてみると、各 LLM で 1 回の処理をしたときのスコア（正解率）は、GPT-4o が 91.5%、Claude 3.5 Sonnet が 89.6%、Gemini 1.5 Pro が 83.5%だった。

それぞれ上記の公表されているスコアと比べると、Claude 3.5 Sonnet が公表値よりも低いが、GPT-4o と Gemini 1.5 Pro は、ほぼ一致している。LLM の出力は、一定ではなく、毎回少しずつ変化するため、公表値とのずれは、LLM のパラメータ設定の違いやサンプル数によるものと考えられる。HumanEval の GitHub リポジトリ[7]では、サンプル数は 200 に設定されているが、今回は、各 LLM の性能ではなく、不正解の問題を比較することが目的のため、サンプル数 1 で処理を実行している。

不正解だった問題は表 1 に示すとおり、GPT-4o が 14 問、Claude 3.5 Sonnet が 17 問、Gemini 1.5 Pro が 27 問である。

表 1　各 LLM が不正解だった問題（不正解は 1，正解は 0）
網掛けは 2 つ以上の LLM が不正解、太字は 3 つの LLM が不正解

問題番号	関数名	GPT-4o	Claude 3.5 Sonnet	Gemini 1.5 Pro
9	rolling_max	0	0	1
10	make_palindrome	0	0	1
26	remove_duplicates	0	0	1
32	find_zero	0	1	1
41	car_race_collision	1	0	0
65	circular_shift	1	1	0
68	pluck	0	0	1
72	will_it_fly	0	1	0
75	is_multiply_prime	1	1	0
77	iscube	0	0	1
83	starts_one_ends	1	0	1
84	solve	0	0	1
86	anti_shuffle	0	1	0
93	encode	1	0	0
102	choose_num	0	1	0
106	f	0	1	0
109	move_one_ball	0	1	0
115	**max_fill**	**1**	**1**	**1**
119	match_parens	0	1	1
120	maximum	0	0	1
121	solution	0	0	1
125	**split_words**	**1**	**1**	**1**
126	is_sorted	0	0	1
127	intersection	1	1	0
129	minPath	1	0	1
130	tri	0	0	1
132	**is_nested**	**1**	**1**	**1**
133	sum_squares	0	0	1
134	**check_if_last_char_is_a_letter**	**1**	**1**	**1**
137	compare_one	0	0	1

138	is_equal_to_sum_even	0	1	0
139	special_factorial	0	0	1
140	fix_spaces	1	0	0
142	sum_squares	0	0	1
145	**order_by_points**	**1**	**1**	**1**
149	sorted_list_sum	0	0	1
160	**do_algebra**	**1**	**1**	**1**
161	solve	0	0	1
163	generate_integers	0	0	1
合計		14	17	27

複数のモデルの利用による正解率の向上

機械学習の分野では、複数のモデルを組み合わせ、より優れた予測精度を実現する手法として、アンサンブル学習がある。単純なアンサンブル手法として、多数決アンサンブルがあり、この考え方を上記の 3 つの LLM に適用する。3 つの LLM のうち、2 つの LLM が正解して、1 つの LLM が不正解の場合、多数決によって、正解を出力できると考える。3 つの LLM を組み合わせれば、表 1 に示すとおり、不正解の問題が 13 問になり、スコア（正解率）は 92.1%に向上する。

また、ソフトウェア開発の現場では、各関数のユニットテストを実行しつつ、エラーが出た場合、コードを修正しながら、開発を進めるのが一般的である。3 つの LLM のうち、いずれかの LLM が正解のコードを出力できれば、コードの修正が可能である。3 つの LLM がすべて不正解だった問題の数は 6 問で、スコア（正解率）は、96.3%に達する。

LLM が正解できなかった問題の傾向

3 つの LLM がすべて不正解だった問題は 6 問あり、その特徴は、関数の docstring に書かれている指示の文章だけではなく、サンプル文から理解する必要がある問題が多いことである。例えば、以下のスクリプトは、134 問目の問題を日本語に翻訳したものである。このサンプル文では、文字列の最後にスペースが入る場合は、False を返す関数を作成するように指定している。

```
def check_if_last_char_is_a_letter(txt):
    '''
    与えられた文字列の最後の文字が以下の条件を満たす場合に True を返し、
    そうでない場合は False を返す関数を作成します：
    1. アルファベットの文字である。
    2. 単語の一部ではない。

    注意：「単語」はスペースで区切られた文字のグループとします。


    例：
    check_if_last_char_is_a_letter(""apple pie"") → False
    check_if_last_char_is_a_letter(""apple pi e"") → True
    check_if_last_char_is_a_letter(""apple pi e "") → False
    check_if_last_char_is_a_letter("""") → False
    '''
```

正解例は次のとおりである。

```
def check_if_last_char_is_a_letter(txt):
    check = txt.split(' ')[-1]
    return True if len(check) == 1 and
    (97 <= ord(check.lower()) <= 122) else False
```

しかし、LLM の回答では、以下のように間違えている。これは GPT-4o の回答だが、Claude 3.5 Sonnet も同じように、最初に右側の空白を削除してしまっているため、正しい判定ができなくなっている。

```
def check_if_last_char_is_a_letter(txt):
    # Strip any trailing whitespace from the string
    txt = txt.rstrip()
    # Check if the string is empty after stripping
    if not txt:
        return False
    # Check if the last character is an alphabetical char
acter

    if txt[-1].isalpha():
        # Split the string by spaces to get the words
        words = txt.split()
        # Check if the last character is a single charact
er word

        if words and words[-1] == txt[-1]:
            return True

    return False
```

一般的なコードの実装では、末尾の空白文字を削除してから、文字列を判定するケースが多くあると考えられ、LLM が同様のコードを学習しているため、確率的に回答していると推測される。

その他、「a を偶数番目、b を奇数番目」にするなど一般的ではない設定がされており、問題文の設定をよく理解する必要がある問題が不正解だった。一般的ではない設定の場合、文章を理解するのが難しい傾向にあるとみられる。

複数のモデルで正解・不正解が割れる問題の傾向

3 つのモデルのうち、Gemini 1.5 Pro が最も間違いが多かったが、以下の問題（65 問目）では、逆に Gemini 1.5 Pro だけが正解だった。

```
def circular_shift(x, shift):
    """
    整数 x の桁を循環的にシフトし、
    桁を右に shift 分だけシフトした結果を文字列として返します。
    shift が桁数よりも大きい場合、桁を逆順にして返します。
    >>> circular_shift(12, 1)
    "21"
    >>> circular_shift(12, 2)
    "12"
    """
```

Gemini 1.5 Pro では、以下のコードに示す通り、shift が桁数より大きい場合のみ逆順にしていて、正解している。

```
def circular_shift(x, shift):
    x = str(x)
    if shift > len(x):
        return x[::-1]
    return x[-shift:] + x[:-shift]
```

GPT-4o では、shift が桁数と同じ場合も逆順にしているため、不正解になっている。

```
def circular_shift(x, shift):
   x_str = str(x)
   num_digits = len(x_str)

   if shift >= num_digits:
       return x_str[::-1]

   shift = shift % num_digits
   return x_str[-shift:] + x_str[:-shift]
```

Claude 3.5 Sonnet でも、shift が桁数と同じ場合も逆順にしているため、不正解になっている。

```
def circular_shift(x, shift):
   # Convert the integer to a string
   x_str = str(x)
   # Get the length of the number
   length = len(x_str)
   # If shift is greater than length, just reverse the s
tring
   if shift >= length:
       return x_str[::-1]
   # Calculate the effective shift (in case shift > leng
th)
   effective_shift = shift % length
   # Perform the circular shift
   shifted = x_str[-effective_shift:] + x_str[:-effectiv
e_shift]
   return shifted
```

この例では、条件分岐の解釈によって、正解と不正解が割れる結果となった。自然言語の問題文からコードを生成する場合、LLM の解釈次第で回答が割れやすいと考えられる。そのため、問題文の書き方をより明確にして、わかりやすいサンプルを多く提示することで、回答の精度が向上すると考えられる。

今回は、3 つの LLM で比較を行ったが、同一の LLM であっても、回答は一定ではないため、複数回の試行のうちの 1 回が正解する確率は、試行回数が多くなるほど、高くなると報告されている[4]。

3 高精度な出力のための手法・技術の調査

生成 AI を活用した効率的なコード生成のために、効果的で高品質な出力を得るための様々な手法や技術が存在する。目的の出力を得るための重要な要素であるプロンプトを中心に、各サービスに共通するベストプラクティスや効果的手法を調査する。

各生成 AI のモデルに共通するベストプラクティス

以下は、LLM を活用した代表的なチャットサービスである ChatGPT[8]、Claude[9,10]、Gemini[11]に共通するプロンプトに関するベストプラクティスの一部である。

- 指示はプロンプトの最初に配置し、指示と文脈を区切るために「###」のような明確な区切り記号を使用する。
- レスポンスを希望するフォーマットを指示する（JSON、XML、markdown など）。
- ペルソナやロールを定義して回答のトーンや語彙を設定する。
- 明確で直接的、かつ詳細な指示を与える。様々な解釈ができるような指示は避ける。
- 問題とその回答の例を提供する。このようにモデルに具体例やデモを提示して文脈的学習を導く手法は Few-shots、 Multi-shots prompt と呼ばれる。
- プロンプトにタスクを解くための推論過程を与える。このような手法は Chain-of-Thought (CoT) と呼ばれる。
- 専門のタスクを担当するそれぞれのエージェントに専門に関連するプロンプトを与える（マルチエージェント）。
- 最終的な答えを一度に出す前に、モデルに返答を考える時間を与える。「ステップバイステップで考えてください」などの指示を含めることで、思考過程を生成させることが可能になる。この手法は Zero-shot CoT と呼ばれる。
- 複雑なタスクをサブタスクに分割する（プロンプト・チェイニング）。
- 「しないでほしいこと」よりも、「してほしいこと」を伝える。
- 知識生成プロンプト - 質問や回答に関連する事前知識を提供する。

ハルシネーションを減らすテクニック

高精度な出力を得るためには、LLM が虚偽の回答をしてしまうハルシネーションを可能な限り回避する必要がある。以下はハルシネーションを減らすテクニックとして各種サービスが紹介するテクニックである。

1. LLM に「わからない」と言うことを許可する。質問や問題への回答が不明な場合や十分な情報がない際に、十分な精度を持つ回答を提示する必要がないことを明示的に指示することで、LLM が不正確な回答を生成する可能性が低くなる。例えば、「自信がない場合、または自信のある答えを提供するのに十分な情報がない場合は、単に「わからない」または「よくわからない」と言ってください」といったプロンプトを含めることである。

2. LLM にタスクを与える際に、目的の出力を提示する前に事前知識やデータを直接引用させてからタスクを実施させる。例えば、タスクを実施する前に、プロンプトに含めた事前知識や具体例などを引用させてからタスクを実施させることで、LLM はそれらのデータを活用しながらタスクに着手する。

3. 生成した出力を次のプロンプトの入力として使用し、入力値である回答内容を検証したり拡張したりする指示を反復的に与える。これにより、LLM が矛盾を発見し、修正する可能性

が向上する。

改良プロンプトを用いて、HumanEval で LLM が正解できなかった問題を再試行

上記の手法・技術を活用しながらプロンプトを改良し、HumanEval において各 LLM が正解できなかった問題の再試行を行った。再試行に用いた問題番号は 115、125、132、134、145、160 である。各 LLM がこれらの問題の正解を導けなかった際のプロンプトは以下である。

```
You are a skilled programmer. Complete the following code snip
pet or function.
```

改良したプロンプトは以下である。また、再試行に利用したモデルは GPT-4o を用いた。

```
You are a world-class problem solver and expert of Pytho
n language tasked with generating high-quality, efficien
t, and accurate code. Your goal is to implement actual f
unction described in the given problem statement using a
function template. Follow these guidelines:

1. Read the problem statement carefully, including any g
iven examples and constraints.
2. Think through the problem step-by-step before writing
 any code.
3. Implement the function exactly as specified, includin
g the function name and parameters.
4. Use appropriate data structures and algorithms to ens
ure efficiency.
5. Include clear and concise comments to explain your lo
gic.
6. Handle edge cases and potential errors carefully.
7. Ensure your code adheres to PEP 8 style guidelines in
 terms of Python coding.
8. After understanding a problem and implement your code
, try to repeatedly verify if the implemented code can a
ctually be an answer for the given problem until you mak
e sure your implementation is correct.
```

その結果、問題 125 split_words において正解となり、プロンプトエンジニアリングによるプロンプトの改良によって一定の効果を示すことがわかった。

4 Sketch2code に関する調査

PBL 活動におけるソフトウェア開発では、初学者もメンバーとして参加していることから、積極的に生成 AI を用いて開発を実施している。生成 AI の利用では主に ChatGPT や Github Copilot などの対話型ツールを使用したが、マルチモーダルに対応しているツールでプロダクトのフロントエンド画面のスクリーンショットを入力としたコード修正なども実施した。こうした背景から、LLM を用いた Sketch2code がフロントエンドの開発に実用的であるかについて精度検証を行うに至った。

Sketch2code はワイヤーフレームを用いたコード（HTML）の自動生成手法であり、手書きやツールで作成したスケッチを基にソースコードを生成することができる。本章ではサンプルのデータセットを用いて Sketch2code の検証を実施した。

Sketch2code での性能評価に関する前提

Sketch2code でのコード生成の性能評価については、標準となっているデータセットがないことから、データサイエンスプラットフォームである Kaggle の sketch2Code データセットを用いて検証した[12]。本データセットは Sketch2code に利用できる様々な種類のワイヤーフレームが格納されており、Kaggle がデータサイエンスの権威的なプラットフォームであるため選定した。

Sketch2code で生成した HTML コードの出力とワイヤーフレームの類似度に関する比較

性能比較については、手書きで構成されたワイヤーフレームを基に生成 AI を用いてコード生成を行い、生成されたコードを基にフロントエンドの画面を表示し、ワイヤーフレームとの類似度を測った。プロンプトとしてはシンプルなものと、前述した高精度な出力のための手法・技術の調査を参考にしたものを利用して、検証のため画像は利用しないように指示をした。

- シンプルなプロンプト

```
添付の画像を元に、HTML を作成してください。ただし HTML 内に画像は利用しないでください
```

- 高精度な出力のための手法・技術の調査を参考にしたプロンプト

```
以下の指示に従って、HTML を作成してください。
- あなたは HTML とウェブデザインの専門家です。
- まず、添付の画像に含まれるすべての要素（テキスト、ボタン、入力フィールドなど）とそのレイアウトを詳細に説明してください。
- 次に、その説明に基づいて、画像と可能な限り一致する HTML コードを作成してください。
- HTML 内では画像ファイルを使用せず、純粋な HTML と CSS のみで再現してください。
- スタイルはインライン CSS ではなく、<style>タグ内に記述してください。
- レスポンシブデザインを考慮し、異なる画面サイズでもレイアウトが崩れないようにしてください。
- 最後に、生成した HTML が画像と一致しているか確認し、必要であれば修正してください。
```

LLM は OpenAI の ChatGPT、Anthropic の Claude、Google の Gemini を使用し、評価軸については、ワイヤーフレームと生成されたコードを基にしたフロントエンド画面の特徴量の比較のため、以下の評価指標を用いた。

- SSIM: 画像の構造的な類似性を評価し、人間の視覚に近い観点から画像の品質や差異を測定。
- PSNR: MSE から計算される信号対雑音比で、画像の品質をデシベル単位で評価。
- Hash Difference: 画像の全体的な視覚的特徴の類似性を評価し、ハッシュ値の差分から画像の類似度を測定。

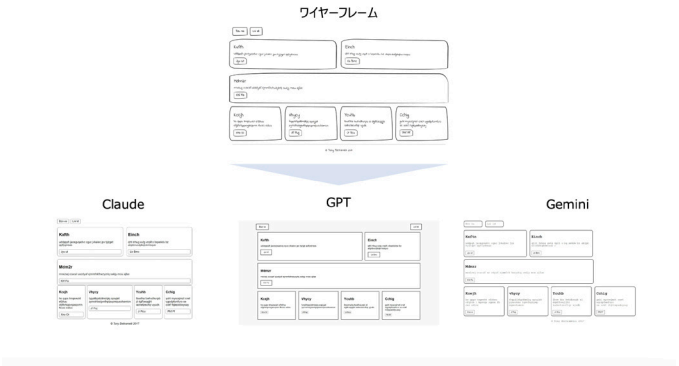


図 2　　ワイヤーフレームを基に各 LLM で生成した HTML

シンプルなプロンプトを使用した比較結果と各 LLM の特性について

今回の検証では 3 つのワイヤーフレームを対象にコード生成を行い、先述した 3 指標を基に類似度を計測した。

ここでは図 2 を対象に、各 LLM の生成物を比較調査した結果を評価する。指標から得た数値を基に、ワイヤーフレームの Sketch2code に有用であると考えられる LLM は Gemini となった

- SSIM: Gemini1.5Pro が最高値（0.788）で、元の画像に最も近いと評価される。
- PSNR: Gemini1.5Pro が最高値（38.41dB）で、ノイズレベルが最も低い。
- Hash Difference: Gemini1.5Pro が最小値（15）で、全体的な視覚的特徴が最も類似している。

表 2　　各 LLM で生成した HTML の類似度比較結果

指標	GPT-4o	Gemini 1.5Pro	Claude3.5 Sonnet
SSIM	0.774	0.788	0.782
PSNR	34.618	38.409	38.194
Hash Difference	16	15	26

他の 2 つのワイヤーフレームも同様に比較したところ、最も有用であると考えられる LLM は Gemini、Claude、GPT の順となった。

Gemini と Claude で生成された出力結果は、作動するボタンやフッターなどワイヤーフレームを忠実に再現されており納得のいくスコアであるが、定性的な評価軸で確認すると Gemini の作成物は上部が左に片寄せされており、矛盾する点がある結果となってしまった。比較結果の数値も Hash Difference を除けば、各 LLM で大きな差はないことからも、どのモデルも一定の水準を保っていると考えられる。

高精度な出力のための手法・技術の調査を参考にしたプロンプトでの再検証

シンプルなプロンプトでの評価と比較をするために、高精度な出力のための手法・技術の調査を参考にしたプロンプトでの検証も実施した。ここでは、シンプルなプロンプトで検証した際に最も精度が高かった、Gemini1.5Pro で再検証した際の結果を記載する。

表 3　　Gemini1.5 Pro での再検証結果

指標	シンプルなプロンプト	改良したプロンプト	差分
SSIM	0.788	0.773	-0.014
PSNR	38.409	37.945	-0.464
Hash Difference	15	20	+5

- SSIM の微減: SSIM が約 0.014 減少し、構造的な類似度がわずかに低下している。
- PSNR の減少: PSNR が約 0.46dB 減少し、画像品質がわずかに低下している。
- Hash Difference の増加: Hash Difference が 5 増加し、視覚的特徴の類似度が低下している。

当初の想定では、高精度な出力のための手法を基にプロンプトを作成した場合はより良い結果が出る想定であったが、結果としてプロンプトを改良したことで、HTML の出力品質が低下した。一方で、GPT-4o と Claude3.5 Sonnet を使用した再検証においては、シンプルなプロンプトでの検証と比較して、改善が見られ数値的にも良い結果を出力することができた。結果として、Gemini1.5 Pro については改良が見られず、GPT-4o と Claude3.5 Sonnet については改良が見られた。真因については、本検証の過程では解明できなかったが、LLM の特性を加味した上でプロンプトを作成することが重要であると認識できた。

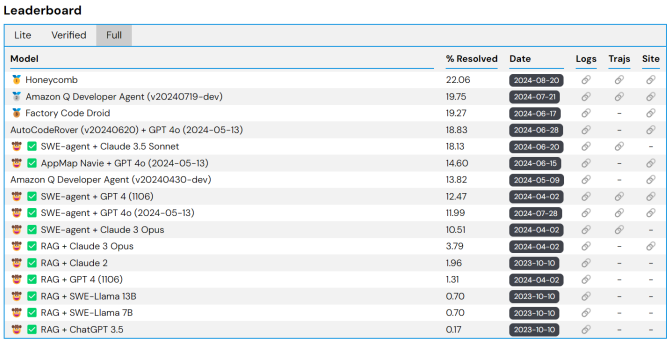
フロントエンド開発における LLM の有用性

原則としてワイヤーフレームを基にした Sketch2code は HTML 形式のフロントエンド成果物を作成するものであり、昨今のライブラリを使用した複雑なロジックが組み込まれたフロントエンドには不向きなものであると考えられる。一方でプロトタイピングなどの簡易的なフロントエンド開発であれば、Sketch2code を用いて早急に開発ができるという点で優れていると考えられる。

5　ソフトウェア開発に即した新しいベンチマークでの評価

ソフトウェア開発での実践的なコード生成能力を評価するため、SWE-bench（Software Engineering Benchmark）という新しいベンチマークの導入が提案されている[13]。SWE-bench では、LLM は、複数の関数、クラス、ファイルにわたる変更を同時に理解して調整する必要があり、非常に長いコンテキストを処理し、複雑な推論を実行する必要がある。SWE-bench の発表論文[14]では、Claude 2 でも 1.96% しか解決できなかったと述べられている。

2024 年 9 月現在の SWE-bench のリーダーボード[13]によれば、図 3 のとおり、20%前後までスコアが向上していることがわかる。上位のモデルには、ソフトウェアの修正、デバッグ、整理に必要なツールを自律的に操作する AI エージェントが並ぶ。AI エージェントは、人間の細かい指示がなくても、高度で複雑な処理を自律的に実行できることが特徴である。AI エージェントを使わない RAG+GPT4 では、スコア 1.31%だったものが、プリンストン大学が開発した SWE-agent[15]を使う SWE-agent+GPT4 では、12.47%にまで向上している。


Leaderboard

Lite Verified Full

Model	% Resolved	Date	Logs	Trajs	Site
Honeycomb	22.06	2024-08-20			
Amazon Q Developer Agent (v20240719-dev)	19.75	2024-07-21			
Factory Code Droid	19.27	2024-06-17		-	
AutoCodeRover (v20240620) + GPT 4o (2024-05-13)	18.83	2024-06-28		-	
SWE-agent + Claude 3.5 Sonnet	18.13	2024-06-20			-
AppMap Navie + GPT 4o (2024-05-13)	14.60	2024-06-15		-	
Amazon Q Developer Agent (v20240430-dev)	13.82	2024-05-09		-	
SWE-agent + GPT 4 (1106)	12.47	2024-04-02			
SWE-agent + GPT 4o (2024-05-13)	11.99	2024-07-28			
SWE-agent + Claude 3 Opus	10.51	2024-04-02			-
RAG + Claude 3 Opus	3.79	2024-04-02		-	
RAG + Claude 2	1.96	2023-10-10		-	-
RAG + GPT 4 (1106)	1.31	2024-04-02		-	-
RAG + SWE-Llama 13B	0.70	2023-10-10		-	-
RAG + SWE-Llama 7B	0.70	2023-10-10		-	-
RAG + ChatGPT 3.5	0.17	2023-10-10		-	-

図 3 SWE-bench のリーダーボード（2024 年 10 月 11 日現在）

AI エージェントを活用したコード生成

AI エージェントを活用すれば、人間が細かい指示を出さなくても、AI が自律的にソフトウェアの新しい機能を実装することができる。SWE-bench でスコア 19.75%を記録している Amazon Q Developer Agent[16]は、コードエディタ Visual Studio Code[17]の拡張機能として利用することが可能である。

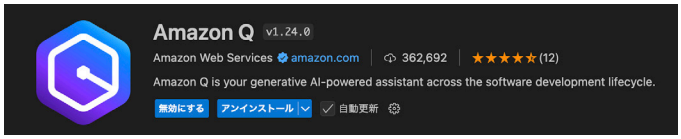


図 4 Visual Studio Code の拡張機能 Amazon Q

例えば、東京の時刻と天気を表示する web アプリを react で実装したい場合、「Build a web app with react that displays the time and weather in Tokyo」とチャットで指示をするだけで、図 5 のとおり、依存関係、複数のコンポーネントが提案され、数分でディレクトリやファイルが生成される。生成されたコードに、気象情報を取得できる API を登録するだけで、実際に web アプリを起動させることができ、図 6 の画面が表示される。

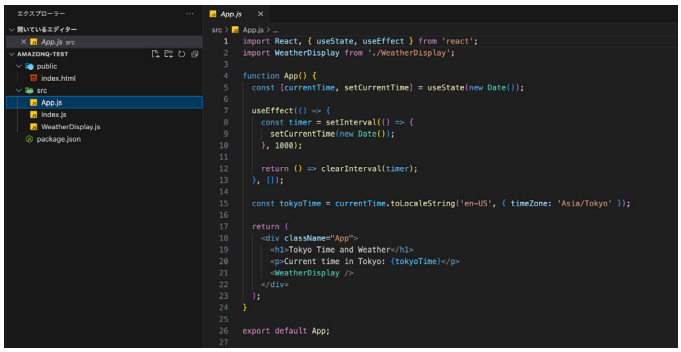


図 5 Amazon Q で生成したコード（複数のディレクトリ、ファイル）

図 6 Amazon Q で実装した天気と時刻を表示する Web アプリ画面

6 おわりに

本紀要では、生成 AI を活用したコード生成の精度と効率化に焦点を当て、最新の大規模言語モデル（LLM）の性能評価と実用性の検証を行った。HumanEval データセットを用いた評価では、GPT-4o、Claude 3.5 Sonnet、Gemini 1.5 Pro といった最新の LLM が 80%以上の高い正解率を示し、複数のモデルを組み合わせることで 96.3%の正解率を達成した。これにより、生成 AI がソフトウェア開発において非常に有用なツールとなり得ることが示された。

一方で、LLM が苦手とする問題の傾向も明らかとなった。特に、複雑な文脈理解を要する問題では正解率が低下する傾向が見られた。これは、プロンプトエンジニアリングの重要性を示唆しており、効果的なプロンプトの作成が生成 AI の性能を最大限に引き出す鍵であることがわかった。

Sketch2code の検証では、ワイヤーフレームから HTML コードを生成する能力を評価し、各 LLM の特性を比較した。結果として、Gemini 1.5 Pro が最も高い性能を示したが、各モデル間の差は僅かであり、いずれも一定の水準を満たしていることが確認された。これは、フロントエンド開発の初期段階やプロトタイピングにおいて、生成 AI が有効に活用できる可能性を示している。

さらに、SWE-bench という新しいベンチマークを用いた評価では、AI エージェントを活用することで、より複雑なソフトウェア開発タスクにおいても生成 AI の有用性が確認された。特に、Amazon Q Developer Agent などの AI エージェントは、人間の細かい指示なしに高度な処理を自律的に実行できる点で注目に値する。

これらの結果は、生成 AI がソフトウェア開発プロセスを大きく変革する可能性を示している。特に、コーディングの効率化やプロトタイピングの迅速化、複雑な開発タスクの自動化などの面で、生成 AI は開発者にとって強力な支援ツールとなり得る。

しかし、課題も残されている。LLM のハルシネーション（誤った情報生成）の問題や、複雑な文脈理解の難しさは依然として克服すべき課題である。また、生成 AI の出力結果を適切に評価し、必要に応じて修正を加える人間の専門知識は引き続き重要である。

今後の展望として、より高度な AI エージェントの開発や、特定のドメインに特化した LLM、人間と AI のより効果的な協働モデルの構築が期待される。また、生成 AI を活用したソフトウェア開発の新しい方法論やベストプラクティスの確立も重要な研究テーマになると考える。

本紀要を通じて、生成 AI がソフトウェア開発の未来に大きな可能性をもたらすことが明らかになったと共に、その潜在力を最大限に引き出すためには、技術の進歩と並行して、倫理的な配慮や人間の創造性との適切な融合が不可欠であることが再認識された。今後も継続的な研究と実践を通じて、生成 AI とソフトウェア開発の関係性をさらに深く探求していく必要がある。

参考文献

1. OpenAI. GPT-4o System Card. 8 Aug 2024 [cited 9 Oct 2024]. Available: https://openai.com/index/gpt-4o-system-card/
2. Anthropic. Claude 3.5 Sonnet Model Card Addendum. [cited 9 Oct 2024]. Available: https://www-cdn.anthropic.com/fed9cc193a14b84131812372d8d5857f8f304c52/Model_Card_Claude_3_Addendum.pdf
3. Gemini Team, Georgiev P, Lei VI, Burnell R, Bai L, Gulati A, et al. Gemini 1.5: Unlocking multimodal understanding across millions of tokens of context. arXiv [cs.CL]. 2024. Available: http://arxiv.org/abs/2403.05530
4. Chen M, Tworek J, Jun H, Yuan Q, Pinto HP de O, Kaplan J, et al. Evaluating large language models trained on code. arXiv [cs.LG]. 2021. Available: http://arxiv.org/abs/2107.03374
5. OpenAI. Hello GPT-4o. 13 May 2024 [cited 9 Oct 2024]. Available: https://openai.com/index/hello-gpt-4o
6. Papers with code - HumanEval benchmark (code generation). [cited 9 Oct 2024]. Available: https://paperswithcode.com/sota/code-generation-on-humaneval
7. OpenAI. human-eval: Code for the paper “Evaluating Large Language Models Trained on Code.” Github; Available: https://github.com/openai/human-eval
8. Best practices for prompt engineering with the OpenAI API. [cited 15 Oct 2024]. Available: https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api
9. Prompt engineering techniques and best practices: Learn by doing with Anthropic’s Claude 3 on Amazon Bedrock. In: Amazon Web Services [Internet]. 3 Jul 2024 [cited 15 Oct 2024]. Available: https://aws.amazon.com/blogs/machine-learning/prompt-engineering-techniques-and-best-practices-learn-by-doing-with-anthropics-claude-3-on-amazon-bedrock/
10. Prompt engineering overview. In: Anthropic [Internet]. [cited 15 Oct 2024]. Available: https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/overview
11. Introduction to prompting. In: Google Cloud [Internet]. [cited 18 Oct 2024]. Available: https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/introduction-prompt-design?authuser=1
12. ShantamVijayputra. Sketch2Code. 2022. Available: https://www.kaggle.com/vshantam/sketch2code
13. Carlos E. Jimenez*, John Yang*, Alexander Wettig, Shunyu Yao, Kexin Pei, Ofir Press, Karthik Narasimhan. SWE-bench. [cited 9 Oct 2024]. Available: https://www.swebench.com/
14. Jimenez CE, Yang J, Wettig A, Yao S, Pei K, Ofir Press, et al. SWE-bench: Can language models resolve real-world GitHub issues? arXiv [cs.CL]. 2023. Available: https://arxiv.org/abs/2310.06770
15. Yang J, Jimenez CE, Wettig A, Lieret K, Yao S, Narasimhan K, et al. SWE-agent: Agent-Computer Interfaces Enable Automated Software Engineering. arXiv [cs.SE]. 2024. Available: https://arxiv.org/abs/2405.15793
16. AWS. Amazon Q Developer, now generally available, includes previews of new capabilities to reimagine developer experience. [cited 9 Oct 2024]. Available: https://aws.amazon.com/jp/blogs/aws/amazon-q-developer-now-generally-available-includes-new-capabilities-to-reimagine-developer-experience/
17. Extensions LMA. Visual Studio Code - code editing. Redefined. [cited 9 Oct 2024]. Available: https://code.visualstudio.com/