Code generation with LLMs

Generative AI && software engineering: analysis, learnings, practical insights

Josh Payne
Agenda

- Intro
- Brief history of AI for code generation
- Benchmarking code gen performance
- Applications and agents
- AI x software engineering
👋 I’m Josh

Founder of Coframe (AI for UI optimization + code gen), prev two other companies (one AI-focused)

Created GPT-Migrate (LLM-powered codebase migration), Coffee (LLM-powered UI code gen)

Stanford CS (AI) alum!
**Brief History**

1. **Source Code (CP):**
   ```java
   public int TextWidth(string text) {
     TextBlock t = new TextBlock();
     t.Text = text;
     return (int) Math.Ceiling(t.ActualWidth);
   }
   ``
   
   **Descriptions:**
   - a. Get rendered width of string rounded up to the nearest integer.
   - b. Compute the actual textwidth inside a textblock.

2. **Source Code (CP):**
   ```java
   var input = "Hello";
   var regex = new Regex("World");
   return !regex.IsMatch(input);
   ``
   
   **Descriptions:**
   - a. Return if the input doesn't contain a particular word in it.
   - b. Lookup a substring in a string using regex.

**Pre-LLM era:**
- RNNs and search

**Early applications:**
- GPT-3, Codex, GitHub Copilot

**"Oh wow, AI can actually write code now":**
- GPT-3.5, GPT-4, OSS LLMs

**AI x software engineering:**
- Agents and integrated workflows

**CodeNN (Iyer et al., 2016)**
- Code summarization

**AROMA Code Recommendation with Extra Lines Highlighted**

```java
public boolean contains(String value) {
    for (String entry : set) {
        if (entry.equalsIgnoreCase(value))
            return true;
    }
    return false;
}
```

**Code search (early copilot)**
- Code search (early copilot)

**Aroma (Luan et al, 2019)**
- Better code summarization

**Code2Seq (Alon et al., 2019)**
- (Try it! -> https://code2seq.org/)
Brief History

Pre-LLM era: RNNs and search

Early applications: GPT-3, Codex, GitHub Copilot

“Oh wow, AI can actually write code now”: GPT-3.5, GPT-4, OSS LLMs

AI x software engineering: Agents and integrated workflows

The world’s most widely adopted AI developer tool.

Get started with Copilot

When the rocket is clicked, temporarily display some text saying “Firing thrusters!”
Brief History

Pre-LLM era:
RNNs and search

Early applications:
GPT-3, Codex, GitHub Copilot

“Oh wow, AI can actually write code now”:
GPT-3.5, GPT-4, OSS LLMs

AI x software engineering:
Agents and integrated workflows
Brief History

Pre-LLM era:
RNNs and search

Early applications:
GPT-3, Codex, GitHub Copilot

“Oh wow, AI can actually write code now”:
GPT-3.5, GPT-4, OSS LLMs

AI x software engineering:
Agents and integrated workflows

Still in its infancy!
Benchmarking code generation

![Graph showing benchmarking of code generation over time with points and lines connecting them. The graph includes labels for Codex-128, CODE-T (code-davinci-002), Parsel (GPT-4 + CodeT), and Reflection (GPT-4). The highest point is for Language Agent Tree Search (GPT-4): 94.400.]
How do we measure this?

1 Benchmark Tasks
2 Competitions
3 Real-world impact

HumanEval (Chen et al., 2021) is the most widely-recognized research benchmark for code generation.

This paper also introduced Codex, the first major code-specific LLM.

HumanEval is 164 handwritten programming problems, each with several unit tests.

```python
def incr_list(l: list):
    """Return list with elements incremented by 1."""
    >>> incr_list([1, 2, 3])
    [2, 3, 4]
    >>> incr_list([5, 3, 5, 2, 3, 3, 9, 0, 123])
    [6, 4, 6, 3, 4, 4, 10, 1, 124]
    return [i + 1 for i in l]
```

The prompt provided to the model is shown with a black background, and a successful model-generated completion is shown in a blue background. To be successful, it must pass the unit tests.
How do we measure this?

1. Benchmark Tasks

There have also been extensions of HumanEval and other datasets:

- **MultiPL-E** is a dataset for evaluating large language models for code generation that supports 18 programming languages. It translates HumanEval problems into other languages.

- **HumanEval-X** consists of 820 high-quality human-crafted data samples, compared with HumanEval's 164.

- **MBPP** (Mostly Basic Python Problems) is a dataset of 1000 crowd-sourced Python programming problems.
How do we measure this?

1. Benchmark Tasks

Some companies will create internal datasets on which to evaluate.

- Google introduced Gemini alongside a new benchmark, Natural2Code, which is a held-out internal dataset.

  GPT-4 (OpenAI) was slightly better on HumanEval (OpenAI), while Gemini (Google) was slightly better on Natural2Code (Google).

- Meta has internal unit test sets for its internal LLMs.

2. Competitions

3. Real-world impact
How do we measure this?

1. Benchmark Tasks
2. Competitions
3. Real-world impact

Why are held-out (non-published) benchmarks valuable?
How do we measure this?

1. Benchmark Tasks
2. Competitions
3. Real-world impact

AlphaCode by DeepMind (Li et al., Dec 2022) created CodeContests, a dataset of compiled competitive programming problems.

Increasingly, datasets from real-world tasks for humans are needed as models approach human-level performance.

Other examples: the LSAT, USMLE, AlphaGeometry (IMO problems)

CodeContests

CodeContests is a competitive programming dataset for machine-learning. This dataset was used when training AlphaCode. AlphaCode has been published in *Science*, with a preprint on arXiv.

It consists of programming problems, from a variety of sources:

<table>
<thead>
<tr>
<th>Site</th>
<th>URL</th>
<th>Source</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aizu</td>
<td><a href="https://judge.u-aizu.ac.jp">https://judge.u-aizu.ac.jp</a></td>
<td>CodeNet</td>
</tr>
<tr>
<td>AtCoder</td>
<td><a href="https://atcoder.jp">https://atcoder.jp</a></td>
<td>CodeNet</td>
</tr>
<tr>
<td>CodeChef</td>
<td><a href="https://www.codechef.com">https://www.codechef.com</a></td>
<td>description2code</td>
</tr>
<tr>
<td>Codeforces</td>
<td><a href="https://codeforces.com">https://codeforces.com</a></td>
<td>description2code and Codeforces</td>
</tr>
<tr>
<td>HackerEarth</td>
<td><a href="https://www.hackerearth.com">https://www.hackerearth.com</a></td>
<td>description2code</td>
</tr>
</tbody>
</table>
How do we measure this?

1. Benchmark Tasks
2. Competitions
3. Real-world impact
How do we measure this?

1. Benchmark Tasks
2. Competitions
3. Real-world impact

As models begin to surpass human performance, they will be increasingly measured on impact.

Example: AlphaDev (Mankowitz and Michi, June 2023) discovered a faster sorting algorithm for small lists that has now been implemented in the C++ standard lib.

SWE KPIs (bug rate, PRs merged, etc) are starting to become more commonplace.
Benchmarking code generation

Benchmark: HumanEval

Techniques

Fine-tuning / Instruct-tuning

Base models
Benchmarking code generation

**Benchmark: HumanEval**

**Base Models** are the GPTs and Llamas of the world: not fine-tuned for a particular task.

**Open LLMs**
- Weights are open, easy to do custom tuning and experimentation
  - CodeLlama (WizardCoder)
  - StarCoder
  - Replit-code-v1-3b
  - Mixtral-8×7b

**Closed LLMs**
- Weights are closed, tuning and experimentation are limited
  - GPT-4
  - Gemini Ultra
  - Claude 2.1
  - Grok
Benchmarking code generation

**Benchmark: HumanEval**

**Base Models** are the GPTs and Llamas of the world: not fine-tuned for a particular task.
Benchmarking code generation

**Benchmark: HumanEval**

**Instruct-tuned models** are models that are fine-tuned with instructions: in this case, for code.
Benchmarking code generation

**Benchmark: HumanEval**

**Instruct-tuned models** are models that are fine-tuned with instructions: in this case, for code.

Instruct-tuning involves a prompt which contains an instruction, and a response. Including the instruction is important for the model to know how to understand new instructions at inference time.

**Example: Synthesis**

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    
    Check if in given list of numbers, are any two numbers closer to each other than given threshold.

    >>> has_close_elements([1.0, 2.0, 3.0], 0.5)
    False
    >>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
    True

    for idx, elem in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx != idx2:
                distance = abs(elem - elem2)
                if distance < threshold:
                    return True

    return False
```
Instruct-tuned models are models that are fine-tuned with instructions: in this case, for code.

Instruct-tuning involves a prompt which contains an instruction, and a response. Including the instruction is important for the model to know how to understand new instructions at inference time.

Example: Fix a bug
Benchmarking code generation

Benchmark: HumanEval

**Instruct-tuned models** are models that are fine-tuned with instructions: in this case, for code.

Instruct-tuning involves a prompt which contains an instruction, and a response. Including the instruction is important for the model to know how to understand new instructions at inference time.

**Example:** Explain code

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    for idx1, elem1 in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx1 != idx2:  # distance = abs(elem1 - elem2)
                if distance < threshold:
                    return True
    return False
```

Provide a concise natural language description of the function using at most 213 characters.

```
Check if in given list of numbers, are any two numbers closer to each other than given threshold.
>>> has_close_elements([1.0, 2.0, 3.0], 0.5)
False
>>> has_close_elements([1.0, 2.8, 3.0, 4.0, 5.0, 2.0], 0.3)
True
```

Write functional code in Python according to the description.

```python
from typing import List

def has_close_elements(numbers: List[float], threshold: float) -> bool:
    for idx1, elem1 in enumerate(numbers):
        for idx2, elem2 in enumerate(numbers):
            if idx1 != idx2:
                distance = abs(elem1 - elem2)
                if distance < threshold:
                    return True
    return False
```
Benchmarking code generation

Benchmark: HumanEval

Instruct-tuning is clearly useful. How can we scale it up?

As LLMs and datasets get larger, we increasingly need to think creatively about how to gather data in order to improve.

One example of this is COMMITPACK: 4 terabytes of Git commits across 350 programming languages (Muennighoff et al, Jan 2024; ICLR preprint).

Git commits naturally pair code changes with human instructions.
Benchmarking code generation

Benchmark: HumanEval

Technique can make all the difference. This is broadly broken down into **reasoning methods** and **decision-making methods**.

<table>
<thead>
<tr>
<th>Technique</th>
<th>Model</th>
<th>Parameters</th>
<th>Score</th>
</tr>
</thead>
<tbody>
<tr>
<td>LATS</td>
<td>GPT-3.5</td>
<td>175B</td>
<td>83.8</td>
</tr>
<tr>
<td>None</td>
<td>GPT-4</td>
<td>1.7T (est)</td>
<td>79.3</td>
</tr>
</tbody>
</table>

💡 GPT-3.5 beats GPT-4 with LATS, despite being 10x smaller!
Chain-of-Thought prompts LLMs to sequentially generate reasoning steps from input to output. It was first introduced in PaLM: Scaling Language Modeling with Pathways. (Chowdhery, Catasta et al., 2022)

However, it suffers from error propagation as the chain length increases.
Benchmarking code generation

Benchmark: HumanEval

Tree-of-Thoughts

Tree-of-Thoughts (ToT) extends CoT by exploring multiple reasoning paths using search algorithms like BFS and DFS. (Yao et al., May 2023)

That said, it is limited by relying solely on the LLM's internal knowledge.
Benchmarking code generation

Benchmark: HumanEval

Reasoning via Planning

Reasoning via Planning (RAP) (Hao et al., October 2023) uses Monte Carlo Tree Search for planning chains of reasoning.

However, it also lacks external feedback.
Benchmarking code generation

**Benchmark: HumanEval**

**ReAct**  
Decision-making method

ReAct prompts LLMs with alternating actions and observations for decision-making in interactive environments. (Yao et al., March 2023)

However, it greedily follows one trajectory and cannot adapt.
Benchmarking code generation

Benchmark: HumanEval

**Reflexion**

*Decision-making method*

Reflexion adds self-reflection to ReAct. This improves overall performance by allowing the LLM more time to think through the problem, similar to CoT. (Shinn et al., October 2023)

However, it does not consider alternative options at each step.
Benchmarking code generation

Language Agent Tree Search

LATS unifies the strengths of both reasoning and decision-making methods through principled search, while overcoming limitations via environmental feedback and self-reflection. (Zhou et al., December 2023)

GPT-4 + LATS is the current best performer on the HumanEval benchmark, with a score of 94.4.
Applications and agents

SOFTWARE IS EATING THE WORLD, BUT AI IS GOING TO EAT SOFTWARE

Jensen Huang / Nvidia CEO
Applications and agents

AI has tackled every aspect of software engineering. (Category list below not exhaustive.)

- replit
- gptengineer
- Coffee by coframe
- Project creation
- SECOND
- GPT-Migrate
- Migrations
- Cody by coframe
- Issues & tests
- IDE
- Cursor
- warp
- Docs
- Dosu
- Sweep AI
- codium
- Mintlify
- docify
Applications and agents

AI has tackled every aspect of software engineering.
(Category list below not exhaustive.)
Applications and agents
AI has tackled every aspect of software engineering. (Category list below not exhaustive.)
Applications and agents

AI has tackled every aspect of software engineering.
(Category list below not exhaustive.)
Applications and agents

AI has tackled every aspect of software engineering.

(Category list below not exhaustive.)

- replit
- gptengineer
- Coffee by codename
- Project creation
- SECOND
- GPT-Migrate
- Migrations
- /c.
- Cody by codename
- IDE
- warp
- Issues & tests
- Docs
- Dosu
- Swim
- Sweep AI
- Mintlify
- Codium
- Docify
Applications and agents
Deep dive: GPT-Migrate

GPT-Migrate
Python -> JavaScript
Applications and agents
Deep dive: Coffee
AI x Software Engineering

- Using code generation wisely
- Prompt engineering for code gen
- AI-driven development
AI x Software Engineering

Using code generation wisely

Credit to Joshua Morony
AI x Software Engineering

Using code generation wisely

Credit to Joshua Morony
AI x Software Engineering

Using code generation wisely

Credit to Joshua Morony
AI x Software Engineering

Using code generation wisely

Credit to Joshua Morony
AI x Software Engineering

Using code generation wisely

Credit to Joshua Morony
AI x Software Engineering

Using code generation wisely

Credit to Joshua Morony
AI x Software Engineering

Using code generation wisely

Why?

- Learning is important
- Understanding your code is important
- Maintainability and knowledge transfer is important
  - Fully LLM-written projects tend to produce “spaghetti code”. I know first-hand!
Prompt engineering is likely more important to code generation than it is to any other area due to the precision required. Luckily, engineers are naturally good prompt engineers.

Principled Instructions are All You Need gives 26 general prompting guidelines (see chart).

Worth adding: only add the minimum viable context; context windows aren’t all made equal.
Prompt engineering for code gen

AlphaCodium formalized “Flow Engineering” for software engineering workflows, which many practitioners had been using already. Using GPT-4 on the CodeContests validation set, the pass@5 accuracy improved from 19% with a well-crafted single prompt to 44% with AlphaCodium.

Code Generation with AlphaCodium: From Prompt Engineering to Flow Engineering

Tal Ridnik, David Keim, Sam Friedman, Carl Waldspurger

Abstract

Code generation problems differ from common natural language problems - they require matching the exact syntax of the target language, identifying input fields and their types, understanding the problem, and addressing other code-specific issues and requirements. Hence, many of the optimizations and ideals that have been successful in natural language generation may not be effective for code tasks. In this work, we present a new approach to code generation by LLMs, which we call AlphaCodium: a test-based, ontology-driven code-oriented iteration flow. This improves the performance of LLMs on code problems. We extend AlphaCodium on a challenging code generation dataset called CodeContests, which includes competitive programming problems, to achieve state-of-the-art results. We show that AlphaCodium improves pass@5 accuracy and improve pass@5 results. On the validation set, for example, GPT-4 accuracy pass@5 increased from 19% to 33% with a single exchanged prompt to 44% with the AlphaCodium flow. Key to this approach is using the principle and best practices acquired in this work, and it can be broadly applicable to general code generation tasks.

Full implementation is available at: https://github.com/alphacodium/alphacodium

1. Introduction

With a space severely rate, code generation tasks require matching the large structural space of possible programs. Current solutions to this problem can be both significantly different and lack a partial or incorrect solution to a problem in a different domain - a single-choice edit on completely alter the solution's behavior. Due to the unique nature of code generation tasks, common programming techniques that have been optimized for natural language code [23, 24, 25, 26] may not be as effective when applied in LLMs.

Figure 1: Flow of Code Generation with AlphaCodium. AlphaCodium uses a two-phase process: pre-processing and code generation. The pre-processing phase involves analyzing and understanding the problem, generating additional data, and setting up the problem for code generation. The code generation phase involves using GPT-4 to generate code and evaluating the generated code on a public test set.

References

[23] have successfully generated code that solves simple programming tasks [1]. However, real-world code problems are often different in nature - they are more nuanced, and not as easily defined with natural language text description. This makes it challenging to design an approach that can achieve competitive performance.

The iteration of CodeContests [14], a dataset created from competitive programming platforms such as Codeforces [11], enables the evaluation of models and their ability to handle these challenging problems, which typically include a single prompt description. A private test set, with more than 3500 test cases for per problem, enables to measure the generated code comprehensively and to ensure that the model can solve the problem.

The primary work addressing the CodeContests dataset was AlphaCodium [1], a code generation system designed by this work. AlphaCodium extends AlphaCodium on a challenging code generation dataset called CodeContests, which includes competitive programming problems, to achieve state-of-the-art results. We show that AlphaCodium improves pass@5 accuracy and improve pass@5 results. On the validation set, for example, GPT-4 accuracy pass@5 increased from 19% to 33% with a single exchanged prompt to 44% with the AlphaCodium flow.
Prompt engineering for code gen

Prompt composition can become complex when you’re dealing with code-writing agents performing multiple types of software engineering tasks.

One solution is organizing them into a hierarchy and creating a constructor that can compose these prompts together, along with any variables you need to pass in from your code.

The simplest way to do this is using text files in labeled directories in your /prompts/ directory. I’m sure there will be headless prompt CMS’s at some point.

Prompt hierarchy in GPT-Migrate
AI x Software Engineering

Prompt engineering for code gen

Sudolang is a natural language constraint-based programming pseudolanguage, with an LLM as the interpreter. What?

More simply, it combines natural language elements and simple coding conventions for better prompting.

SudoLang prompts can often be written with 20% - 30% fewer tokens than natural language.

The expressiveness and precision helps when writing code, as well as when “programming” the LLM to serve as an application itself.
AI x Software Engineering

AI-driven development: practical pointers
Language preference

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.
AI x Software Engineering

AI-driven development: practical pointers

**Language preference**

LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

**Project structure**

Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.
**AI x Software Engineering**

Al-driven development: practical pointers

<table>
<thead>
<tr>
<th>Language preference</th>
<th>Project structure</th>
<th>Interface-oriented programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLMs do better with more popular languages. They also benefit from the clarity of typed languages.</td>
<td>Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.</td>
<td>LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.</td>
</tr>
</tbody>
</table>
AI x Software Engineering

Language preference
LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

Project structure
Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.

Logs-in-the-loop
When debugging (or in a background loop), LLMs can digest logs and error traces. Very helpful!

Interface-oriented programming
LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.
AI x Software Engineering

AI-driven development: practical pointers

**Language preference**
LLMs do better with more popular languages. They also benefit from the clarity of typed languages.

**Project structure**
Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.

**Logs-in-the-loop**
When debugging (or in a background loop), LLMs can digest logs and error traces. Very helpful!

**Tests, tests, tests**
When generating entire functions and files, test coverage is CRUCIAL. (LLMs can write these too!)

**Interface-oriented programming**
LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.
## AI x Software Engineering

**AI-driven development: practical pointers**

<table>
<thead>
<tr>
<th>Language preference</th>
<th>Project structure</th>
<th>Interface-oriented programming</th>
</tr>
</thead>
<tbody>
<tr>
<td>LLMs do better with more popular languages. They also benefit from the clarity of typed languages.</td>
<td>Try to keep files and modular. Use headers and TDDs to help the LLM navigate and generate files.</td>
<td>LLMs need context. Interfaces (input, output, transformation, types) give this. Use IOP in prompts.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Logs-in-the-loop</th>
<th>Tests, tests, tests</th>
<th>Output structure</th>
</tr>
</thead>
<tbody>
<tr>
<td>When debugging (or in a background loop), LLMs can digest logs and error traces. Very helpful!</td>
<td>When generating entire functions and files, test coverage is CRUCIAL. (LLMs can write these too!)</td>
<td>YAML uses as little as 50% of the tokens that JSON output does. Even with JSON mode, YAML wins.</td>
</tr>
</tbody>
</table>
Acknowledgements

- Michele Catasta
- Pavlo Razumovskyi
- Glavin Wiechert
- Tinah Hong
- Alex Korshuk
- John Whaley

Thank you!
Questions