# An In-Depth Investigation On Current LLM Code Generation And Feasibility Of LLM-Based Code Generation In Software Development

Pranav Hariharane
UNI: pkh2120

**Abstract: In this paper, I convey the most recent research conducted on LLM-based code generation and highlight if there's potential for LLMs to achieve good performance in generating code for software development. I also discuss limitations in traditional code generation benchmarks, new benchmarks created for the purpose of evaluating LLMs in software development code scenarios, and recent techniques explored to improve the code generation performance for LLMs. Finally, I replicate an experiment from one of the papers I reviewed to confirm its results and conclude with an evaluation of the implications that it has on the potential for LLMs to produce realistic software development code in the future.**

## I. Introduction

Recently, there have been a lot of studies on how Large Language Models(LLMs) can be used to generate code based on a natural language description. When pre-trained on massive amounts of natural language documentation and code, LLMs like Codex, CodeLlama, and GPT-4 have demonstrated the capability to generate decent quality programs across many different programming languages. This indicates possibilities for LLMs to be used in aiding developers with their daily software development tasks and improving their overall productivity. Despite recent progress, it's still unclear how feasible it may be for LLMs to generate good quality code for pragmatic software development since LLM-based generated code has proven to often contain errors including security vulnerabilities, syntax errors, difficult to pinpoint bugs, etc. Most studies only focus on evaluating LLM-based generated code on simple standalone functions that aren't representative of the code required for real-world software development and the potential issues introduced. Thus, through my research, I aim to shed light on exactly how LLMs currently perform in synthesizing code that is applicable to pragmatic software development use cases. I also aim to provide some insight into how the accuracy, correctness, and usability of the code generated by LLMs can be improved through both prompting techniques and additional context provided from test cases. By addressing these gaps in current studies, I hope to provide insights into the future work required in LLM-based code generation so that it can be used to help developers with tasks and problems brought up in real-world software development.

### *A. What the reader will learn*

In this paper I highlight the limitations of the current benchmarks used to evaluate LLM-based code generation and present an overview of the recent benchmarks that have been developed to address these limitations. I also present recent efforts to improve the performance of LLMs in code generation through different prompting techniques and different ways of utilizing test-cases as additional context to provide LLMs with. After reading this paper, I hope readers will have a much clearer understanding about the current performance, limitations, and areas of improvement or further study required for LLM-based code generation to be reliably used to aid software developers.

## II. Background

In order to better understand the content of this paper, there are a few concepts that I need to explain. First of all, the term benchmark will be commonly used throughout this paper. Benchmarks are simply just datasets used to evaluate the code generation performance of LLMs. For each coding problem, most benchmarks typically provide a natural language description of the problem, the function signature, the ground-truth solution, and test cases. Upon evaluation, LLMs usually get prompted with the natural language description and the function signature to generate code and the generated code gets evaluated on the test cases. If the generated code passes all the test cases by producing outputs equivalent to the ground truth solution, the generated code is considered correct. The most common code generation benchmarks are HumanEval(164 manually created coding problems that support around 18 programming languages), MBPP(around 1,000 Python coding problems), and APPS(around 10,000 Python coding problems). The main limitation with these benchmarks, which this paper will discuss in more detail later, is that they only include simple function-level coding tasks. This makes it difficult to evaluate exactly how well LLMs fare in software development code generation tasks.

When evaluating LLM-based generated code, the most common metrics are BLEU, CodeBLEU, and Pass@k. BLEU is basically a word and phrase similarity checking metric which scores LLM-based generated code based on how similar it is to the ground truth solutions for the problems. CodeBLEU is an improvement upon BLEU that also takes the generated code structure into account when comparing it with the ground truth solution. Pass@k is a percentage metric that represents the probability that at least one solution out of k generated solutions passes all the test cases provided. The articles I present in the Literature Review section mainly use the Pass@k metric since the correctness of the generated code is much more important than the similarity that generated code has with the ground truth solution.

All LLMs have a temperature configuration which is a metric indicating how deterministic or creative the responses from the LLM will be. Temperature ranges from 0 to 1 with higher values resulting in more creative and diverse outputs from the LLM. Another major concept that will be mentioned throughout the paper is prompting. Prompting refers to the general process of providing input to an LLM in a way that helps it learn certain behaviors or functionality. There's three general forms of prompting: zero-shot prompting, few-shot prompting, and chain-of-thought prompting. Zero-shot prompting refers to when an LLM is asked to perform a task without any examples or guidance provided in the input. Few–shot prompting refers to when an LLM is provided with a few examples in the input before being asked to generate a response. Chain-of-thought prompting requests an LLM to reason about the input task step-by-step before generating a response.

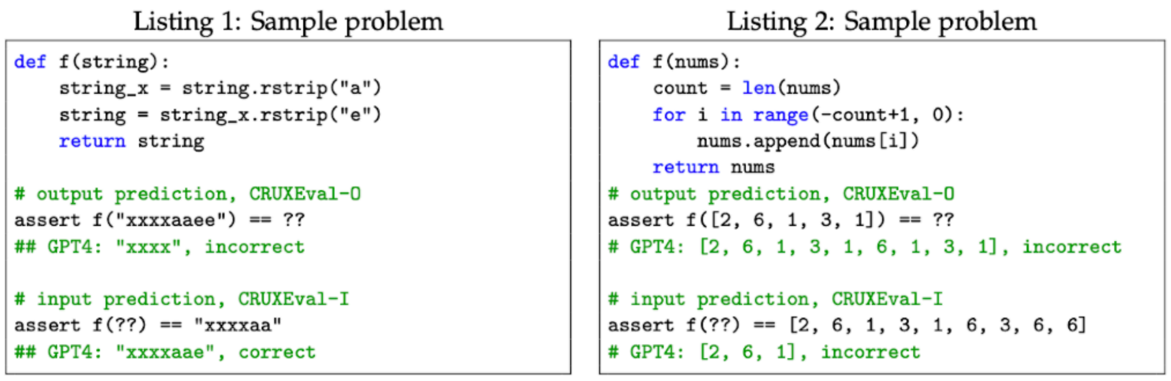
## III. Literature Review

### *A. Recent Evaluation Benchmarks for Code Generation*

Recently, there has been a lot of research dedicated to understanding exactly how well LLMs are able to generate code from a natural language description. It turns out that most of the commonly used benchmarks(e.g. HumanEval, MBPP, etc.) are actually weak and inadequate for assessing the performance of LLM-based generated code. The current widely used benchmarks provide less than 10 tests on average for each coding problem. On top of that, these tests are usually too simple to explore the full functionality of the code and cover edge cases. These benchmarks also don’t provide clear natural language problem descriptions for many coding problems leading to many capable LLMs misjudging the intent of the problem. The descriptions provided in these benchmarks are often too vague and don’t fully clarify the expected behavior of the program. Even the ground-truth solutions for the coding problems in these benchmarks have been found to contain errors. These insufficiencies have led to a lot of research in creating code generation benchmarks that can properly measure the performance of LLM-based generated code.

One popular framework that was developed in 2023 is EvalPlus [5]. EvalPlus aims to augment existing code generation benchmarks by generating tests that fully exercise the functionality of the generated code. At a high level, EvalPlus first uses ChatGPT to generate a set of high-quality tests that fully cover the generated code along with corner cases. Using these high-quality tests, EvalPlus performs type-aware mutation to efficiently generate a large number of additional test cases. These newly generated tests are then used to evaluate the code generated by LLMs against the ground-truth implementations for the corresponding coding tasks. The authors of [5] use EvalPlus to create new benchmarks(HumanEval$^+$ and HumanEval$^+$-Mini where HumanEval$^+$-Mini is a subset of HumanEval$^+$ that still fulfills the same testing requirements). They evaluated a variety of code generation LLMs based on these new benchmarks and found that the scores produced by the LLMs on HumanEval$^+$ were consistently lower than the scores produced on HumanEval since HumanEval$^+$ was able to detect incorrect code solutions misidentified as correct by HumanEval. The LLMs even produced a different performance ranking when tested on HumanEval$^+$ [5] further confirming how HumanEval is an inadequate benchmark for measuring the performance of LLM-based generated code.

Another popular benchmark developed in 2024 that aims to improve upon the existing code generation benchmarks is CRUXEval [7]. Traditional code generation benchmarks aim to just capture the ability of LLMs to generate proper code for simple tasks. However, there’s an absence of benchmarks that aim to measure the capabilities of LLMs in code understanding and execution. CRUXEval aims to measure the performance of LLMs in reasoning about code execution behavior by evaluating LLMs on two tasks: input prediction behavior(predict the input for a given function based on the provided output) and output prediction behavior(predict the output for a given function based on the provided input). Figure 1 illustrates examples of these two tasks [7].



Listing 1: Sample problem

```python
def f(string):
    string_x = string.rstrip("a")
    string = string_x.rstrip("e")
    return string

# output prediction, CRUXEval-O
assert f("xxxxaaee") == ??
## GPT4: "xxxx", incorrect

# input prediction, CRUXEval-I
assert f(??) == "xxxxaa"
## GPT4: "xxxxaae", correct
```

Listing 2: Sample problem

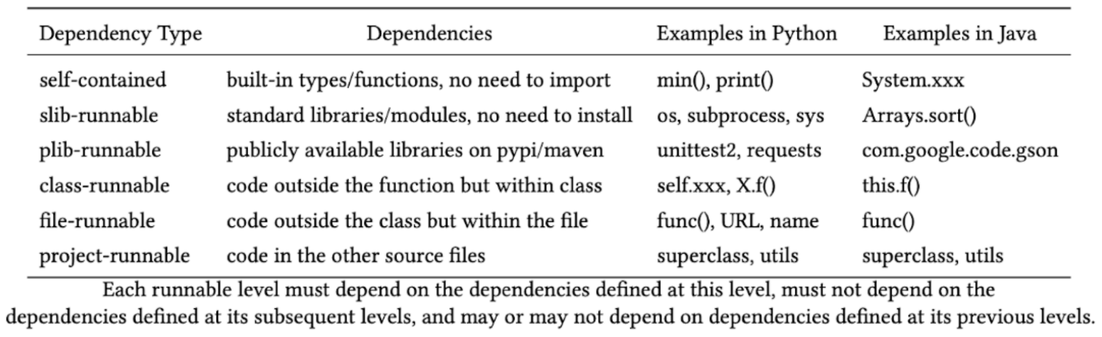
```python
def f(nums):
    count = len(nums)
    for i in range(-count+1, 0):
        nums.append(nums[i])
    return nums
# output prediction, CRUXEval-O
assert f([2, 6, 1, 3, 1]) == ??
# GPT4: [2, 6, 1, 3, 1, 6, 1, 3, 1], incorrect

# input prediction, CRUXEval-I
assert f(??) == [2, 6, 1, 3, 1, 6, 3, 6, 6]
# GPT4: [2, 6, 1], incorrect
```

Figure 1. Example Input/Output Prediction Tasks [7]

The authors created the CRUXEval benchmark with 800 Python functions that each come with an input-output pair to evaluate LLMs on these new tasks. The authors also tested the traditional HumanEval benchmark and this newly created benchmark on 20 code generation LLMs and saw that stronger CRUXEval performance generally correlates with stronger HumanEval performance [7]. This implies that the ability of LLMs to reason about code plays a major role in their ability to generate good quality code. CRUXEval was also able to reveal a significant gap between GPT-4 and smaller models. Despite its stronger performance, GPT-4 failed to understand the behavior of some very simple Python programs that the smaller models were able to grasp easily [7]. The authors even explored the effects of fine-tuning and chain-of-thought prompting in LLMs for these input/output prediction tasks and found that it significantly improves their CRUXEval performance [7]. Overall, the CRUXEval benchmark contributed by [7] serves as a good starting point in understanding the code reasoning abilities of LLMs and future work still needs to be done to measure the impact of these capabilities with other programming languages, code corpuses with higher difficulty, various other prompting techniques, etc.

Most benchmarks used in recent times only contain simple standalone function-level coding tasks which are unrepresentative of realistic software development scenarios. As a result, there has been some research exploring the creation of benchmarks that can handle pragmatic software development coding tasks. One of these benchmarks is CoderEval [6]. The authors of the CoderEval paper created the benchmark by selecting high-quality functions from the most popular Java and Python open-source projects on Github. The functions are classified based on their dependencies(commonly referred to as runnable levels in the CoderEval paper) which fall into these 6 categories illustrated in Table 1 [7].



| Dependency Type | Dependencies | Examples in Python | Examples in Java |
|---|---|---|---|
| self-contained | built-in types/functions, no need to import | min(), print() | System.xxx |
| slib-runnable | standard libraries/modules, no need to install | os, subprocess, sys | Arrays.sort() |
| plib-runnable | publicly available libraries on pypi/maven | unittest2, requests | com.google.code.gson |
| class-runnable | code outside the function but within class | self.xxx, X.f() | this.f() |
| file-runnable | code outside the class but within the file | func(), URL, name | func() |
| project-runnable | code in the other source files | superclass, utils | superclass, utils |

Each runnable level must depend on the dependencies defined at this level, must not depend on the dependencies defined at its subsequent levels, and may or may not depend on dependencies defined at its previous levels.

Table 1. Dependency Types [7]

Each function is categorized based on its runnable level. The hierarchy of runnable levels is illustrated in Figure 2 [6].

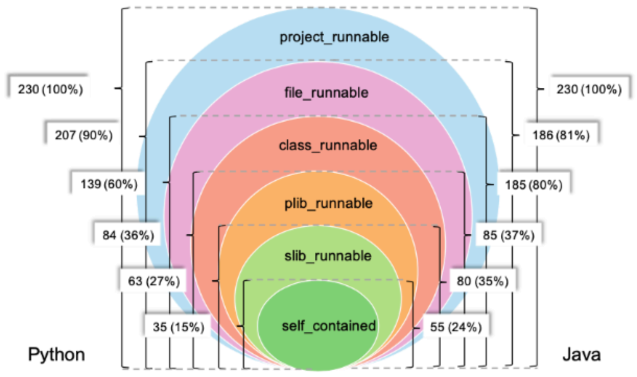
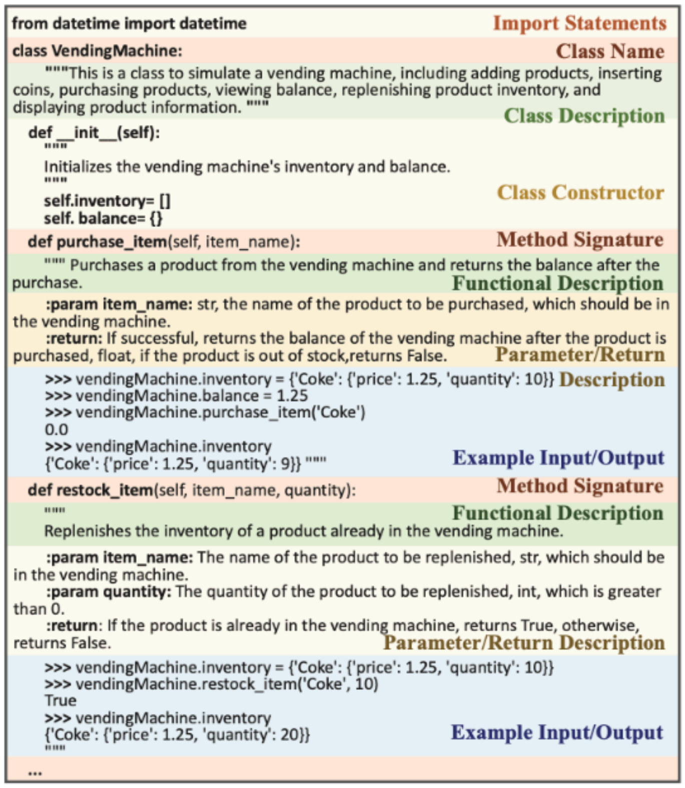


Figure 2. Distribution of Runnable Levels in CoderEval

The authors evaluated 3 different code generation LLMs on this benchmark and compared the results with the HumanEval benchmark. The external dependencies for the coding problems in the CoderEval benchmark were supplied in the prompts to ensure that the LLMs had sufficient information to generate the corresponding code. Unsurprisingly, all 3 LLMs displayed significantly better performance on the HumanEval benchmark confirming how LLMs are currently only capable of generating code for simple standalone functions [7]. The authors even evaluated how well the models incorporated the external dependencies they were prompted with for each coding task in CoderEval and found that the ability the LLMs showed in including the external dependencies correlated with their ability to generate the correct code [7]. This indicates that more effort needs to be directed towards figuring out how to improve the ability of LLMs in both understanding and being able to utilize external dependencies when generating code for non-standalone functions.

Another recent benchmark developed in 2024, which aims to address the evaluation of LLMs on code generation for realistic software development scenarios, is ClassEval [1]. ClassEval contains 100 Python class-level code generation tasks that can evaluate an LLM's capability of generating a class with multiple interdependent methods and how well the generated code captures contextual dependencies [1]. Each code generation task in the ClassEval benchmark comprises a description for the target class, a test suite, and a ground truth solution. Figure 3 displays an example of what the description for the target class might look like.



```
from datetime import datetime
class VendingMachine:
    """This is a class to simulate a vending machine, including adding products, inserting
coins, purchasing products, viewing balance, replenishing product inventory, and
displaying product information. """
  def __init__(self):
      """
      Initializes the vending machine's inventory and balance.
      """
      self.inventory= []
      self. balance= {}
  def purchase_item(self, item_name):
      """ Purchases a product from the vending machine and returns the balance after the
purchase.
      :param item_name: str, the name of the product to be purchased, which should be in
the vending machine.
      :return: If successful, returns the balance of the vending machine after the product is
purchased, float, if the product is out of stock,returns False.
      >>> vendingMachine.inventory = {'Coke': {'price': 1.25, 'quantity': 10}}
      >>> vendingMachine.balance = 1.25
      >>> vendingMachine.purchase_item('Coke')
      0.0
      >>> vendingMachine.inventory
      {'Coke': {'price': 1.25, 'quantity': 9}} """
  def restock_item(self, item_name, quantity):
      """
      Replenishes the inventory of a product already in the vending machine.
      :param item_name: The name of the product to be replenished, str, which should be
in the vending machine.
      :param quantity: The quantity of the product to be replenished, int, which is greater
than 0.
      :return: If the product is already in the vending machine, returns True, otherwise,
returns False.
      >>> vendingMachine.inventory = {'Coke': {'price': 1.25, 'quantity': 10}}
      >>> vendingMachine.restock_item('Coke', 10)
      True
      >>> vendingMachine.inventory
      {'Coke': {'price': 1.25, 'quantity': 20}}
      """
  ...
```

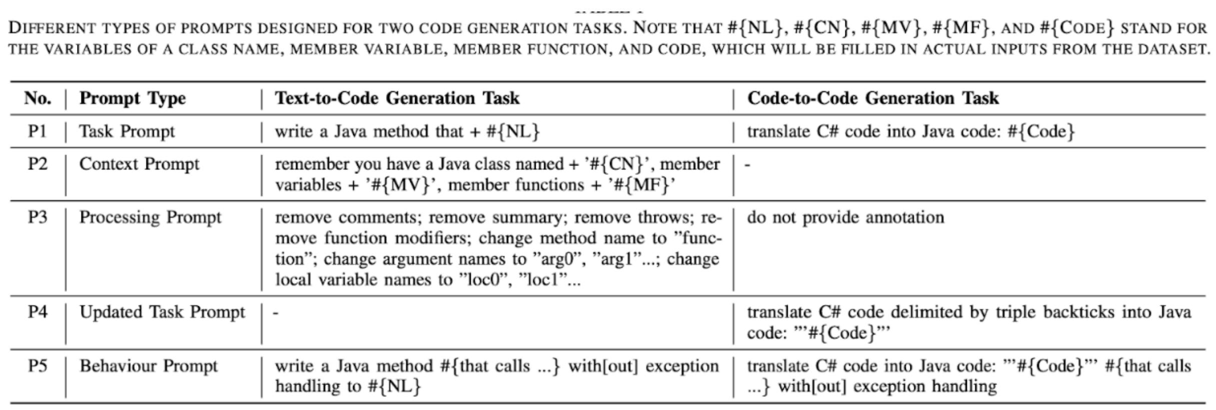
Figure 3. Example Target Class Input Skeleton [1]

The authors evaluated 11 different code generation LLMs on ClassEval and found that the LLMs demonstrate substantially lower performance on class-level code generation tasks compared to standalone method-level code generation tasks [1]. This disparity probably occurs due to the difficulties that these LLMs encounter in understanding the dependencies that each class method has on other class fields and methods. Consequently, more research needs to be conducted to see how we can get LLMs to be aware of the dependencies that exist in pragmatic code.

*B. Prompting Techniques To Improve LLM-Based Code Generation*

Given the results that LLMs have produced on existing and recently developed code generation benchmarks, there have been many efforts dedicated to finding ways to improve the performance of LLMs in code generation. One technique that has proven to be very impactful is prompting. Enhancing the prompts provided to LLMs can significantly improve the quality of the code that it generates.

Recent research conducted in 2023 on ChatGPT explores various prompting strategies and their effect on code generation tasks [4]. The authors of [4] tested different combinations of the 5 prompts illustrated in Table 2.



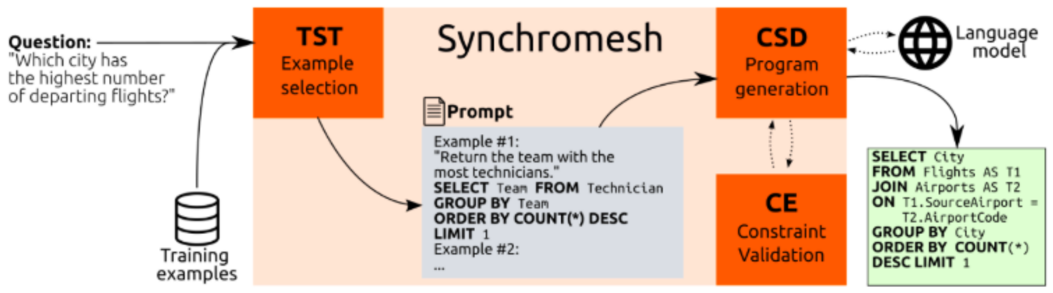
DIFFERENT TYPES OF PROMPTS DESIGNED FOR TWO CODE GENERATION TASKS. NOTE THAT #{NL}, #{CN}, #{MV}, #{MF}, AND #{CODE} STAND FOR THE VARIABLES OF A CLASS NAME, MEMBER VARIABLE, MEMBER FUNCTION, AND CODE, WHICH WILL BE FILLED IN ACTUAL INPUTS FROM THE DATASET.

| No. | Prompt Type | Text-to-Code Generation Task | Code-to-Code Generation Task |
|---|---|---|---|
| P1 | Task Prompt | write a Java method that + #{NL} | translate C# code into Java code: #{Code} |
| P2 | Context Prompt | remember you have a Java class named + '#{CN}', member variables + '#{MV}', member functions + '#{MF}' | - |
| P3 | Processing Prompt | remove comments; remove summary; remove throws; remove function modifiers; change method name to "function"; change argument names to "arg0", "arg1"...; change local variable names to "loc0", "loc1"... | do not provide annotation |
| P4 | Updated Task Prompt | - | translate C# code delimited by triple backticks into Java code: '''#{Code}''' |
| P5 | Behaviour Prompt | write a Java method #{that calls ...} with[out] exception handling to #{NL} | translate C# code into Java code: '''#{Code}''' #{that calls ...} with[out] exception handling |

Table 2. Different Prompt Types Used in [4]

When testing the effect that different reasonable combinations of these prompts had on the CodeXGLUE benchmark, the authors found that the code generated by ChatGPT can improve substantially when guided by more specific and detailed prompts [4]. The authors also analyzed the effect that adding a request for conciseness in the prompt would have and found that it further improved the performance of ChatGPT, generating code that was closer to the ground-truth solution [4]. Although these various prompting methods brought the generated code closer to the canonical solutions, many of the generated solutions still struggled to satisfy complete correctness indicating the need for future efforts in resolving the correctness of LLM-based generated code.

One prompting framework that employs few-shot prompting to combat the logical and implementation issues produced in LLM-based generated code is Synchromesh, which was developed in 2022 [9]. The Synchromesh framework has two components: TST(Target Similarity Tuning) and CSD(Constrained Semantic Decoding). The TST component queries a pre-trained similarity model to retrieve example tasks and solutions relevant to the problem description provided in the input. These example tasks and solutions are provided in a prompt to the underlying LLM which works with the CSD component to iteratively refine the generated code so that it's free of any syntactic or semantic errors enforced by the programming language and input context. Figure 4 illustrates a high-level overview of the framework.



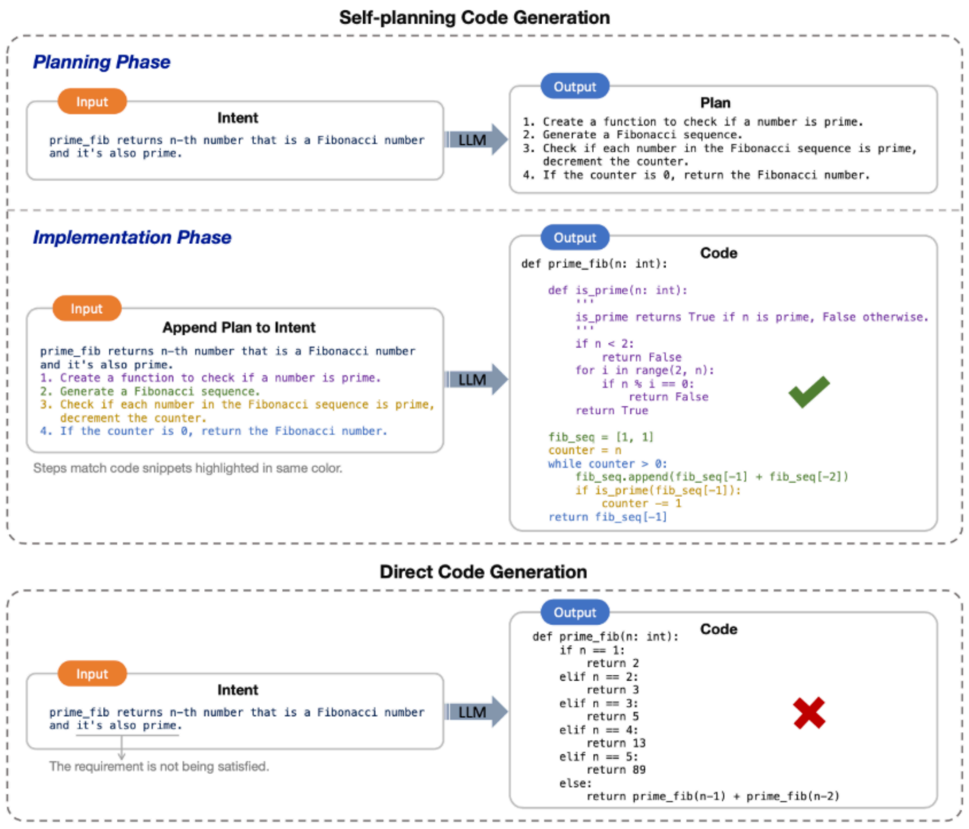
```
Example #1:
"Return the team with the most technicians."
SELECT Team FROM Technician
GROUP BY Team
ORDER BY COUNT(*) DESC
LIMIT 1
Example #2:
...
```

```
SELECT City
FROM Flights AS T1
JOIN Airports AS T2
ON T1.SourceAirport =
   T2.AirportCode
GROUP BY City
ORDER BY COUNT(*)
DESC LIMIT 1
```

Figure 4. High Level Overview of Synchromesh [9]

Upon evaluating the framework with three different code generation LLMs(GPT-3 13B, GPT-3 175B, Codex 175B) and three different programming languages(SQL, SMCalFlow, and Vega-Lite), the authors found that Synchromesh improved the accuracy of the code generated for all the LLMs [9]. Synchromesh also drastically improved the validity of the generated code, preventing it from producing any semantic or syntax errors [9]. At the same time, the framework still struggled to prevent many conceptual errors from appearing in the generated code [9]. Also, it's still unclear how Synchromesh extends to general purpose languages like Python or Java where it's much harder to enforce syntactic and semantic constraints.

Another promising prompting technique that was proposed in 2024 is Self-Planning Code Generation [3]. This approach to prompting LLMs has 2 phases: a planning phase and an implementation phase. In the planning phase, an LLM is prompted with the coding problem description x along with k randomly selected examples concatenated together. These k examples represent <x', y'> pairs where x' represents a coding problem description for a different task and y' represents the corresponding optimal plan(numeric list detailing the steps to generate code for x'). Upon receiving this prompt, the LLM attempts to generate an optimal plan for x which we can call y. In the implementation phase, the LLM is prompted with y and asked to generate code that follows the steps listed in y. Figure 5 illustrates an example showing how these phases work.



```
prime_fib returns n-th number that is a Fibonacci number
and it's also prime.
```

```
1. Create a function to check if a number is prime.
2. Generate a Fibonacci sequence.
3. Check if each number in the Fibonacci sequence is prime,
   decrement the counter.
4. If the counter is 0, return the Fibonacci number.
```

```
prime_fib returns n-th number that is a Fibonacci number
and it's also prime.
1. Create a function to check if a number is prime.
2. Generate a Fibonacci sequence.
3. Check if each number in the Fibonacci sequence is prime,
   decrement the counter.
4. If the counter is 0, return the Fibonacci number.
```

```
def prime_fib(n: int):

    def is_prime(n: int):
        '''
        is_prime returns True if n is prime, False otherwise.
        '''
        if n < 2:
            return False
        for i in range(2, n):
            if n % i == 0:
                return False
        return True

    fib_seq = [1, 1]
    counter = n
    while counter > 0:
        fib_seq.append(fib_seq[-1] + fib_seq[-2])
        if is_prime(fib_seq[-1]):
            counter -= 1
    return fib_seq[-1]
```

```
prime_fib returns n-th number that is a Fibonacci number
and it's also prime.
```

```
def prime_fib(n: int):
    if n == 1:
        return 2
    elif n == 2:
        return 3
    elif n == 3:
        return 5
    elif n == 4:
        return 13
    elif n == 5:
        return 89
    else:
        return prime_fib(n-1) + prime_fib(n-2)
```

Figure 5. Self-Planning Code Generation Example [3]

The authors evaluated this approach on different variants of the HumanEval and MBPP benchmarks with the Code-Davinci-002 LLM and compared the results from the self-planning code generation approach with other traditional techniques like chain-of-thought prompting and few–shot prompting. The self-planning approach demonstrated the best performance in terms of the correctness of the generated code beating all the other baseline techniques [3]. The authors even analyzed the impact of problem complexity and found that the self-planning approach displayed the most significant improvement compared to the baselines in handling higher difficulty problems while also being able to maintain good performance for the simpler tasks [3].

## *B. Including Test Case Context To Improve LLM-Based Code Generation*

Apart from prompting, another technique that has proven to be very useful in improving the performance of LLMs in code generation is providing test cases as additional context. One framework that explores the effect that adding test cases in addition to a coding problem description has on code generation is the TGen framework [10]. The framework pretty much prompts an underlying LLM to generate code for a coding problem repeatedly for a fixed number of iterations until it passes all the provided test cases. When the LLM produces code that fails certain tests, the TGen framework adds the failure information and advice on how to address the failures to the prompt for the next iteration. The authors tested this framework on the MBPP, HumanEval, and CodeChef(dataset with competition-level programming problems) benchmarks with the GPT-4 Turbo LLM. The results showed that the addition of test cases in the prompt and the process of repeatedly prompting the LLM to fix test case errors further boosted the accuracy and correctness of the code generated by the GPT-4 Turbo model [10]. The problems that remained unsolved occurred due to these reasons: the LLM misunderstood the requirements in the problem description, the LLM was unable to follow the advice provided to fix test case errors, and the LLM failed to cover edge cases like empty arrays, negative numbers, etc [10].

Although providing test cases as additional context in the prompt for LLMs does improve their performance in code generation, there’s a lot of manual costly and time-consuming effort involved with creating test cases. To address this, the CodeT framework was developed in 2022 [8]. In its usage, CodeT first accepts a coding problem description and uses an underlying LLM to generate test cases and possible code solutions for the problem. Then CodeT executes each code solution it generates against every test case and ranks each code solution based on the number of test cases it passes and the number of other code solutions that are similar to it. Lastly, CodeT returns the highest ranked solution to the user. Figure 6 illustrates a high-level overview of this process.

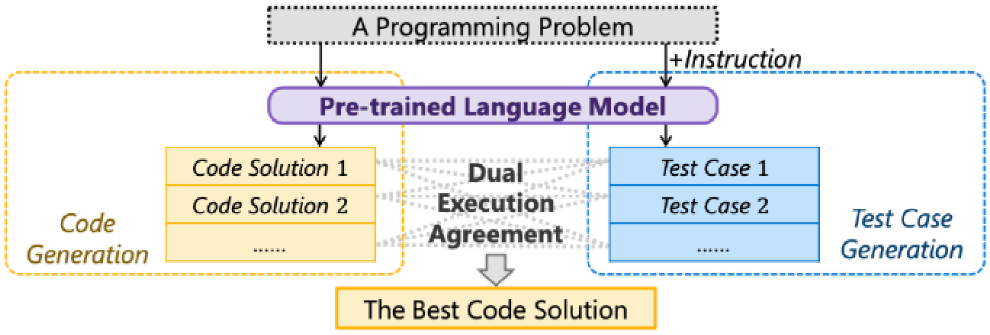
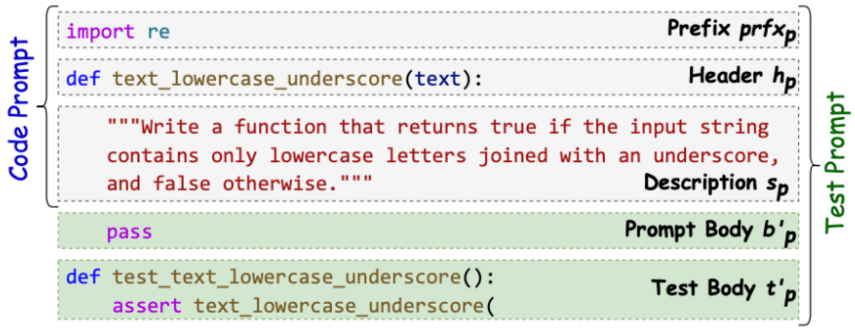
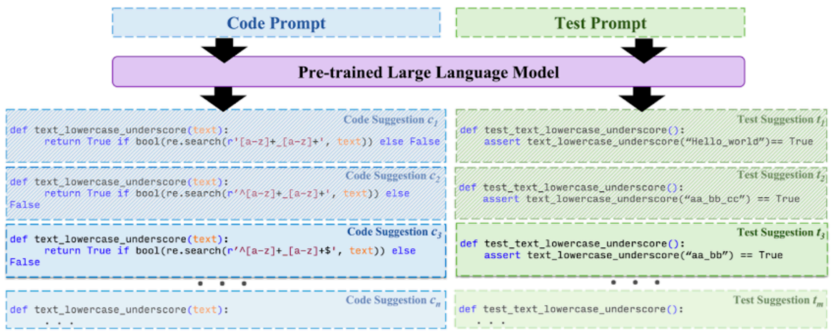


Figure 6. High-Level Overview of CodeT [8]

The authors tested this framework on 4 popular code generation benchmarks with different code generation LLMs and found that CodeT displayed better code generation performance across all models and benchmarks [8]. The authors also found that generating better quality test cases correlated to better code generation performance [8]. This was confirmed when the authors tested the code produced by CodeT when using mediocre LLMs with the test cases produced by a stronger LLM and saw a significant improvement in overall code generation performance. Despite the increased performance observed by the CodeT framework, there were a decent number of instances where a correct solution was generated but not given the highest rank leading to an incorrect solution being returned to the user [8]. This issue most likely occurred due to the LLMs inability to understand certain problem descriptions or address edge cases.

In response to the issue of LLMs misunderstanding coding problem descriptions, the TiCoder framework was developed in 2024 [11]. The TiCoder framework aims to address the ambiguity and informal nature present in many coding problem descriptions by clarifying the user intent through test cases. When used, TiCoder first accepts a coding problem description with optional additional code context. Then it generates a set of candidate code and test case suggestions by prompting the underlying LLM. Figure 7 illustrates an example of what these candidate code and test case suggestions might look like.


```
import re
def text_lowercase_underscore(text):
    """Write a function that returns true if the input string
    contains only lowercase letters joined with an underscore,
    and false otherwise."""
    pass
def test_text_lowercase_underscore():
    assert text_lowercase_underscore(
```



```
def text_lowercase_underscore(text):
    return True if bool(re.search(r'[a-z]+_[a-z]+', text)) else False

def text_lowercase_underscore(text):
    return True if bool(re.search(r'^[a-z]+_[a-z]+', text)) else
False

def text_lowercase_underscore(text):
    return True if bool(re.search(r'^[a-z]+_[a-z]+$', text)) else
False
  • • •
def text_lowercase_underscore(text):
    . . .
```
```
def test_text_lowercase_underscore():
    assert text_lowercase_underscore("Hello_world")== True

def test_text_lowercase_underscore():
    assert text_lowercase_underscore("aa_bb_cc") == True

def test_text_lowercase_underscore():
    assert text_lowercase_underscore("aa_bb") == True
  • • •
def test_text_lowercase_underscore():
  . . .
```

Figure 7. Example Code and Test Case Suggestions Generated by TiCoder Framework [11]

The TiCoder framework then ranks the test cases according to a discriminative policy and presents the user with the test case that discriminates the most among the generated code solutions. The top ranked test is presented as a query to the user who responds to verify whether the test case is consistent with the user intent or not. Depending on the user response, the TiCoder framework may prune the set of generated code solutions before repeating the ranking and user test case validation steps as necessary. At the end, the TiCoder framework outputs a set of generated tests along with a ranked list of code solutions consistent with the user responses. The authors tested the TiCoder framework on the MBPP and HumanEval datasets with a variety of code generation LLMs. Since TiCoder requires real-time user responses, the authors used the ground truth implementations for the coding problems to simulate the user response to whether a test case was consistent with the user intent. The results showed that the TiCoder framework was able to significantly improve the code generation performance for all the LLMs [11]. In addition, as the number of user test validation queries increased, there was consistent improvement in the performance of the generated code [11]. TiCoder was even able to boost the accuracy of smaller LLMs to the point where their performance was comparable to larger LLMs with just 1 user validation query [11].

## *C. Usability and Human Studies on Code Generation Tools*

When evaluating the performance of LLM-based code generation, it's important to not only consider the correctness of the generated code but also the usability since the main purpose of the code generation task is to eventually be able to provide people with code that they can safely and responsibly use. Thus, it's important to also evaluate the experience users have when using LLMs for code generation tasks. Even if the code produced by an LLM is correct, it might not be beneficial if it's hard for users to follow or understand.

The authors of [11] conducted a user study with the TiCoder framework where they asked 15 participants to perform 3 different coding tasks under 3 conditions: directly prompting the underlying LLM with just a coding problem description, using the traditional TiCoder framework, and using a variant of the TiCoder framework where the user can provide the correct output for the user test case validation query. The authors measured the user performance under these 3 conditions with respect to the correctness of the generated code, time required for the users to complete the tasks, and the cognitive load reported by the users. When using the TiCoder framework, the correctness of the generated code was significantly higher, the users required less time to complete the tasks, and the users reported significantly less cognitive load [11].

Another user study conducted in 2022 compared the experience that 24 users had in three real-world Python code generation tasks when using Copilot(a popular code generation tool powered by Codex) versus Intellisense(the default code completion plugin provided in VSCode) [2]. The authors reported the amount of time users took to complete a task, whether the solution produced by the users was correct, and the feedback users gave on their experience when using each tool. Even though most participants preferred using Copilot over Intellisense and even suggested that Copilot was more helpful, users that used Copilot tended to fail to complete certain code generation tasks in the allotted time more often [2]. This mainly occurred when users struggled to understand and debug the incorrect code generated by Copilot, especially if the users didn't have any prior knowledge on how to approach the task. There was also not a significant difference between the time users took to solve code generation tasks with Intellisense and Copilot due to the time that users needed to spend debugging the code generated by Copilot [2]. The biggest benefit of Copilot reported by the users was that it provided a really good starting point for problems where users didn't really know how or where to start [2].

## IV. Experiment

I originally wanted to conduct an experiment where I evaluated one of the prompting techniques or test–driven techniques used to improve code generation LLMs on the ClassEval benchmark to see if I could improve upon the results in [1]. However, I found that all the online code for these enhancement techniques assume that the benchmarks they are evaluated with follow the same format as HumanEval or MBPP. I was unable to find a valid way to integrate these techniques with the way that the input data is formatted for the ClassEval benchmark. As a result, I decided to try and replicate the results from [1] to confirm the results provided in the paper and gain more insight into the ClassEval benchmark.

### *A. Setup*

This is the original link for the ClassEval benchmark Github repository: https://github.com/FudanSELab/ClassEval. I had to make a few modifications to the repository to replicate the experiment on my local machine. The URL for the repository with my changes is https://github.com/hariharanep/ClassEvalCodeGenerationExperiment. The instructions to replicate the process I followed to execute this small experiment on my local machine can be seen in the Github repository README.

For this experiment, I decided to test the top 2 LLMs that performed the best in generating class-level code in [1] and also executed without issues on my local machine. For the first part of my experiment, I configured the LLMs to have a default temperature of 0.2, used a nucleus sampling strategy, and used a holistic generation strategy(LLM is asked to produce the entire code for the class all at once) when prompting the LLMs to generate code. However, I only generated 3 samples for each coding task instead of 5 due to time and resource constraints. I reported the class-level and method-level Pass@k results for each LLM. The class-level Pass@k denotes the probability that at least one entire class was generated correctly out of k classes and the method-level Pass@k denotes the probability that at least one method was generated correctly out of k methods.

For the second part of this experiment, I decided to test the effect that different generation strategies had on the Pass@1 results. The different generation strategies are holistic generation(LLM is asked to produce the entire code for the class all at once), incremental generation(LLM is asked to produce the code for the class in the traditional way starting with the attributes, constructor, one method, more complex methods, etc.), and compositional generation(LLM is asked to produce the entire code for the class method by method). Instead of nucleus sampling, the samples for these results were generated using a greedy decoding algorithm that only generates one sample for each coding task.

### *B. Results*

| Model | Class-Level Pass@1 | Class-Level Pass@3 | Class-Level Pass@5 | Method-Level Pass@1 | Method-Level Pass@3 | Method-Level Pass@5 |
|---|---|---|---|---|---|---|
| GPT-4 | 0.31 | 0.35 | 0.36 | 0.54 | 0.60 | 0.62 |

| | | | | | | |
|---|---|---|---|---|---|---|
| GPT-3.5 | 0.13 | 0.16 | 0.17 | 0.21 | 0.26 | 0.28 |

Table 3. Pass@k with Nucleus Sampling(3 samples created for each problem)

| Model | Holistic | Incremental | Compositional |
|---|---|---|---|
| GPT-4 | 0.32 | 0.22 | 0.27 |
| GPT-3.5 | 0.14 | 0.21 | 0.22 |

Table 4. Effect of Different Generation Strategies(Only Reporting Class-Level Pass@1 Greedy Scores)

*C. Findings*

I was able to execute the code to evaluate the GPT-4 and GPT-3.5 LLMs code generation performance on this benchmark pretty smoothly since the OpenAI APIs available in Python make access to these LLMs relatively easy. Most of the other LLMs gave me a lot of issues since they weren't able to allow me to execute the code with just the CPU available on my Mac. Even the LLMs that did allow me to invoke their usage with just a CPU produced issues related to some of the code being outdated for the specific versions of the libraries that were used.

In the ClassEval paper([1]), the results for the Pass@k values obtained with nucleus sampling for the GPT-4 and GPT-3.5 LLMs followed the same pattern as the results I obtained in this experiment. Both results reinforce how performance on class-code generation tasks is much lower than the performance on method-level code generation tasks for existing LLMs. One major reason for this is that existing LLMs still aren't able to fully understand and be aware of the contextual dependencies that a specific method in a class has. These dependencies include class fields, other class methods, third party libraries, etc. The main difference between the results in [1] and my results were that my results had much lower scores which most likely occurred because I opted to evaluate the LLMs with nucleus sampling where 3 samples were generated for each coding task instead of 5. Since the experiment that [1] conducted was able to generate more samples for each coding task, the probability that at least one of k samples generated is correct should be considerably higher across all reported metrics for both LLMs.

For the comparison across different generation strategies, my experiment produced different results than the results produced in [1]. My results followed the same pattern as the GPT-4 results in [1]. However, the GPT-3.5 results I produced were completely different. In [1], the GPT-3.5 LLM performed best with holistic generation followed by incremental generation and lastly compositional generation. In my experiment, the GPT–3.5 LLM performed best with compositional generation followed by incremental generation and lastly holistic generation. This most likely occurred due to my usage of evaluating the Pass@1 greedy decoding scores instead of the Pass@5 nucleus sampling scores which [1] reported. The results from my experiment are more likely to be random and not as representative of the true effect that the generation strategies have on LLM-based code generation due to the fact that less samples were getting generated for each task. However, I think it's still worthwhile to further explore the effect that the generation strategy has on LLM-based code generation performance since even the results in [1] don't show any statistical significant difference between the performances of each generation strategy.

## V. Conclusion

In this paper, I presented the latest research on LLM-based code generation benchmarks, techniques for improvement, and usability experience. I explained the limitations of current code generation benchmarks(lack of sufficient test cases, ambiguous problem descriptions, lack of problems that are representative of realistic software development scenarios) and proposed frameworks that aim to address these limitations in different ways. I even highlighted recent prompt-driven and test-driven approaches used to improve the code generation performance of LLMs. Although the results from these approaches are promising, they were mainly tested on current code generation benchmarks like HumanEval, MBPP, and other similar variants. So, it's still unclear exactly how much they'll improve LLM-based code generation for software development tasks. Even in the usability studies where developers were tasked with using LLM-based code generation tools to generate solutions to different tasks, bugs were fairly common in incorrect LLM-based generated code and sometimes required significant effort to understand and debug.

The results I received in my experiment reinforce the major limitations of LLM–based code generation as seen by how poorly LLMs are able to generate class-level code compared to method-level code. The fact that class-level code generation isn't even among the most complex software development tasks further proves how far existing LLMs are from being able to produce reliable code for software development tasks. I think further research should be done to explore how we can improve the ability of LLMs to both understand and be aware of contextual dependencies that exist in most realistic software development code. I believe that this is the next major milestone towards getting LLMs to eventually be able to produce reliable software development code.

## VI. What I learned

Before starting my research on this topic, I knew that existing LLMs were only able to perform decent code generation for simple function-level programming problems. However, I wasn't too clear about the reasons for this. After performing this research, I gained a much clearer understanding on why LLMs perform well on method-level functions while performing poorly for most realistic software development tasks. Also, one of my goals for this research paper was to gain some familiarity and experience actually using LLMs. I was able to gain this experience through the small-scale experiment I performed for this paper. Even though I was able to use a lot of code from an existing public repository, I had to take some time to really understand the code in order to make changes to get it to work on my local machine.

## VII. References

[1] Du, X. *et al.* (2024) ‘Evaluating large language models in class-level code generation’, *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pp. 1–13. https://dl.acm.org/doi/10.1145/3597503.3639219

[2] Vaithilingam, P., Zhang, T. and Glassman, E.L. (2022) ‘Expectation vs. experience: Evaluating the usability of code generation tools powered by large language models’, *CHI Conference on Human Factors in Computing Systems Extended Abstracts*, pp. 1–7. https://dl.acm.org/doi/abs/10.1145/3491101.3519665

[3] Jiang, X. *et al.* (2024) ‘Self-planning code generation with large language models’, *ACM Transactions on Software Engineering and Methodology*, 33(7), pp. 1–30. https://arxiv.org/abs/2303.06689

[4] Liu, C. *et al.* (2023) *Improving CHATGPT prompt for code generation*, *arXiv.org*. Available at: https://arxiv.org/abs/2305.08360 (Accessed: 23 February 2025).

[5] Liu, J. *et al.* (2023) *Is your code generated by CHATGPT really correct? rigorous evaluation of large language models for code generation*, *arXiv.org*. Available at: https://arxiv.org/abs/2305.01210 (Accessed: 23 February 2025).

[6] Yu, H. *et al.* (2024) *Codereval: A benchmark of pragmatic code generation with generative pre-trained models*, *arXiv.org*. Available at: https://arxiv.org/abs/2302.00288 (Accessed: 23 February 2025).

[7] Gu, A. *et al.* (2024) *Cruxeval: A benchmark for code reasoning, understanding and execution*, *arXiv.org*. Available at: https://arxiv.org/abs/2401.03065 (Accessed: 23 February 2025).

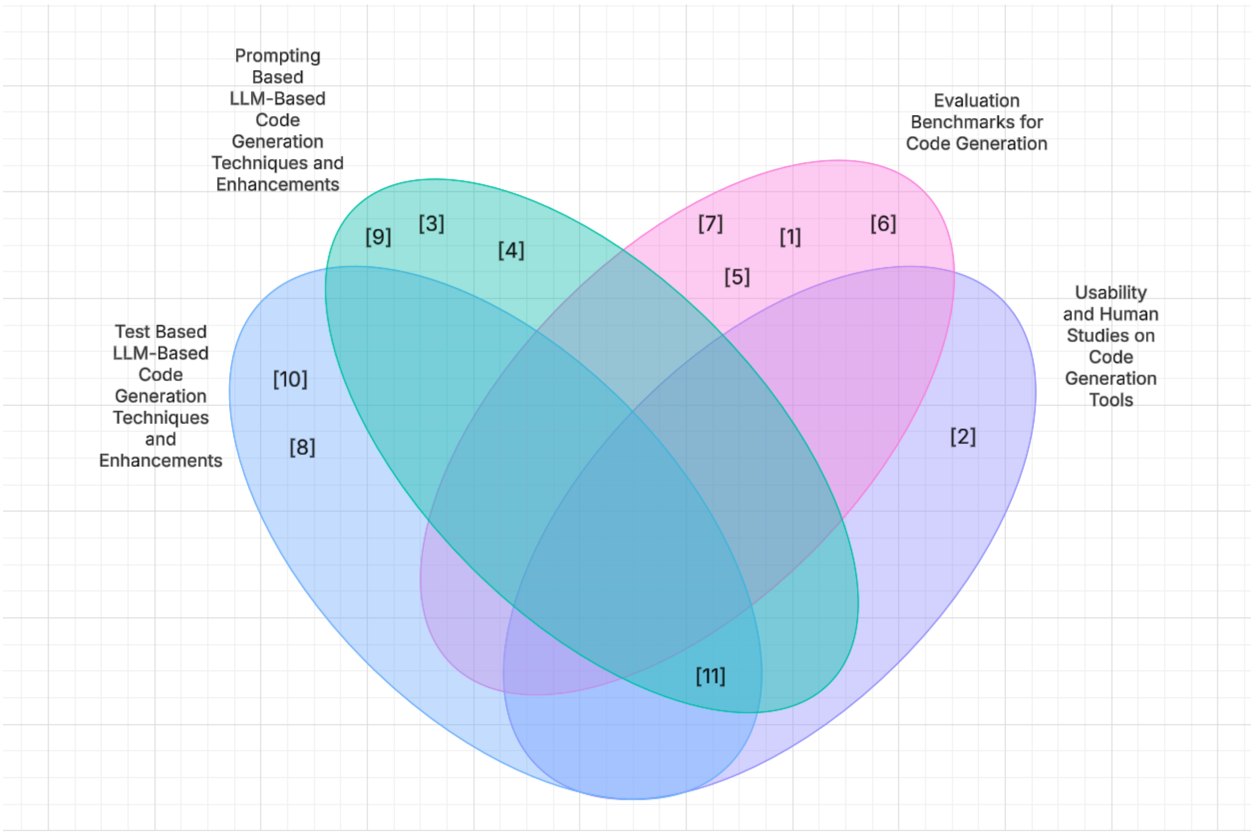
[8] Chen, B. *et al.* (2022) *Codet: Code generation with generated tests*, *arXiv.org*. Available at: https://arxiv.org/abs/2207.10397 (Accessed: 23 February 2025).

[9] Poesia, G. *et al.* (2022) *Synchromesh: Reliable code generation from pre-trained language models*, *arXiv.org*. Available at: https://arxiv.org/abs/2201.11227 (Accessed: 23 February 2025).

[10] Mathews, N.S. and Nagappan, M. (2024) ‘Test-driven development and LLM-based Code Generation’, *Proceedings of the 39th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1583–1594. https://arxiv.org/abs/2402.13521

[11] Fakhoury, S. *et al.* (2024) ‘LLM-Based Test-Driven Interactive CODE GENERATION: User study and empirical evaluation’, *IEEE Transactions on Software Engineering*, 50(9), pp. 2254–2268. https://arxiv.org/abs/2404.10100

## VIII. Appendix


Prompting Based LLM-Based Code Generation Techniques and Enhancements
Evaluation Benchmarks for Code Generation
Usability and Human Studies on Code Generation Tools
Test Based LLM-Based Code Generation Techniques and Enhancements
[9]
[3]
[4]
[7]
[1]
[6]
[5]
[10]
[8]
[2]
[11]