Sample-Efficient LLM-Driven Program Synthesis: A Novel Bayesian Optimization Approach

Abstract—**We consider the task of generating functionally correct code using large language models (LLMs). The correctness of generated code critically depends on the prompt used to query the given base LLM. We formulate the problem of finding the appropriate prompt as combinatorial search process and propose a novel Bayesian optimization (BO) approach referred to as *BO for Code GENeration (BODE-GEN)*. BODE-GEN performs an adaptive data-driven search over prompts guided by training data in the form of prompts tried and the functional accuracy of the generated code over a set of given test cases. The key insight is to perform BO in continuous embedding space by using an auxiliary LLM to bridge the gap between discrete prompt space and continuous embedding space. We leverage two synergistic ideas, namely, random projections and dimensionality scaled priors, to build effective Gaussian process based surrogate models over the high-dimensional embedding space. Our extensive experiments on the HumanEval+ benchmark using multiple base LLMs show that BODE-GEN improves significantly in terms of code generation accuracy compared to fixed prompts and manual prompt engineering. Additionally, we demonstrate that BODE-GEN is sample-efficient, requiring relatively few iterations of BO to achieve substantial gains in code accuracy.**

Index Terms—**AI for software development, large language models, program synthesis, Bayesian optimization**

I. Introduction

Large language models (LLMs) have emerged as transformative tools in various domains, including software development. Their ability to assist with code-related tasks has positioned them as indispensable coding assistants for software developers today [8], [9], [25]. However, as developers increasingly rely on code generated by LLMs, the functional correctness of this code has become a critical factor in ensuring the overall quality of software products.

In modern software development, the software supply chain comprises of various components, including low-level systems software, application frameworks, third-party libraries, and build tools. These components are often inter-dependent and any flaw in one can propagate through the entire system, leading to significant and widespread challenges [11], [12]. When developers incorporate LLM-generated code into these components, the functional correctness of the code becomes vital. Incorrect code generated by LLMs can introduce subtle bugs that are difficult to detect and diagnose, potentially causing cascading side-effects for the entire software supply chain [4]. Therefore, ensuring the correctness of LLM-generated code before its incorporation into the software development process is of paramount importance.

Recent studies have highlighted that LLMs can produce incorrect code, for various reasons, including hallucinations and insufficient understanding of coding tasks [20], [25], [33]. To address this challenge, some potential solutions include better pre-training datasets and improved training / fine-tuning methods to create high-performing LLMs specifically for coding tasks [14]. There is also an inherent trade-off between resource cost and performance of LLMs. On one end of the spectrum, large LLMs require huge amount of high-quality training data and compute resources for both training and inference [13], [36]. On the other end, training and inference with smaller LLMs is feasible and computationally cheap, but these models may struggle to generalize well to handle the complexities of real-world code synthesis [34], [35]. In this context, leveraging large foundational LLMs such as ChatGPT through *prompting* has emerged as a promising avenue for developers seeking to generate high-quality code [7], [19], [23], [24]. However, recent studies have shown that treating LLMs as black-boxes and relying on standard/manual prompting can result in the generation of incorrect code [3], [16], [21], [30], [31].

Improving automated prompting approaches is complementary to alternative approaches of improving the code generation capabilities of LLMs. Prior work in this direction include knowledge augmentation [2], [16] and reasoning elicitation [1], [23]. Enhanced by self-consistency [37], the correctness of generated code repair can be further improved with CoT [1]. However, it is not clear if similar improvement can be achieved for synthesizing programs from scratch. Increasing the specificity of the code-generating prompts can provide additional help [24], but determining the specificity level is currently a manual process.

Test-driven software development [6] is an effective software engineering paradigm where developers write tests before code to write correct code. Inspired by the practical success of this paradigm, this paper asks the following question: *Given a base LLM and test cases for a coding task, can we develop an automated prompting approach to generate correct code by minimizing the number of tried prompts (sample-efficiency)?* There are two key challenges in answering this question. First, the search space of prompts is combinatorial and huge. Second, querying the base LLM with a candidate prompt and evaluating the accuracy of code on test cases is expensive. Therefore, we have a problem akin to finding needles in a big haystack. This paper answers this question by proposing a novel prompt search approach based on the framework of Bayesian optimization (BO) [29]. The key idea behind BO is to learn a surrogate model from the past evaluations (prompt and code accuracy pairs) and use it to intelligently select a sequence of prompts to achieve our goal (generating code with 100 percent accuracy on the given test case).

Our proposed *BO for Code GENeration (BODE-GEN)* approach performs search in continuous embedding space by using an auxiliary LLM to bridge the gap between discrete prompt space and continuous embedding space. The popular Gaussian process (GP) [26] based surrogate models work well in low dimensions (less than 50), but the embedding dimension for auxiliary LLM such as LLama2 is 4096 which poses significant challenges. To address this high-dimensional challenge, we leverage two synergistic ideas, namely, random projections and dimensionality scaled priors to build effective GP based surrogate models which are critical for the success of BO. BODEGEN addresses the core issue of incorrect code generation using LLMs at its source and provides a practical and effective solution for developers. It improves the reliability of LLM-generated code and mitigates the potential cascading side effects in the software supply chain, contributing to the development of more robust and secure software systems.

Contributions. The key contributions of this paper include:

- Development of a novel Bayesian optimization approach (BODE-GEN) to iteratively improve prompts for a given base LLM to solve code synthesis tasks.
- Demonstrating the effectiveness of BODE-GEN in generating code that meets functional requirements with higher correctness (fraction of passed test cases).
- Empirical evidence for BODE-GEN outperforming baseline prompting techniques by a good margin. Our code and data will be made available on a GitHub repo. We provide our anonymous code repository for reviewing purposes: https://anonymous.4open.science/r/BODE-GEN-F2A8
- Identification of good prompting practices that may lead LLMs to generate code with higher correctness.

II. Problem Setup and Challenges

Let LLM_{base} denote a base large language model (LLM) that can be queried using textual prompts to solve coding tasks. Given an initial prompt p_0 (e.g., *"write Python code for testing whether a given number is prime or not"*) for a coding task T and a set of n developer-provided test cases to verify the correctness of the generated code, our goal is to find a prompt $\hat{p}$ that when used with LLM_{base} will generate functionally correct code, i.e., produces correct outputs on all n test cases.

Suppose Accuracy(LLM_{base}, p) represent the the functional accuracy of the code generated by the given base LLM using the prompt p on n test cases (e.g., 90 percent accuracy means the code passes 90 percent of the given n test cases). Our goal is to find a prompt $\hat{p}$ which maximizes this accuracy and ideally achieves 100 percent accuracy. This problem can be mathematically formulated as follows.

$$\hat{p} = \arg\max_{p} \textsc{Accuracy}(LLM, T) \tag{1}$$

Key Challenges. There are two main challenges in solving this optimization problem.

- *Large combinatorial space of prompts.* Each prompt is a sequence of valid tokens. Given a token vocabulary and maximum size of the sequence, the search space of all candidate prompts is combinatorial and very large.
- *Expensive-to-evaluate objective function.* To evaluate the accuracy of a candidate prompt p, we need to query the base LLM to generate code and execute it on all n test cases. Each query to base LLM is expensive in terms of dollar cost or computational cost.

Therefore, our goal is to minimize the number of queries to the objective function (i.e., the number of tried prompts) to solve this optimization problem. Random search and trial-and-error approaches are not compatible with this goal because their exploration strategy doesn't incorporate machine learning and decision-theoretic reasoning to achieve the target goal.

III. Bayesian Optimization for Prompt Search

In this section, we describe a novel approach for sample-efficient prompt search based on the framework of Bayesian optimization (BO). First, we provide the necessary background on BO. Next, we describe the key challenges in using BO for prompt search and our proposed BODE-GEN approach based on continuous embeddings to address those challenges.

A. Background on Bayesian Optimization

BO [29] is a derivative-free method to *adaptively* and *efficiently* search a given input space X (e.g., search space of prompts) to optimize expensive-to-evaluate objective function $f(x \in X)$. BO is an adaptive procedure because it intelligently selects inputs from X by iterating between querying the objective function y=$f(x \in X)$ and making a decision about which input to query next x_{next}. BO is sample-efficient because it makes a data-driven decision to select the next input to query the objective function by taking into account all input-output pairs from previous query evaluations.

Each decision to select the next input from $\mathcal{X}$ to evaluate with f must trade-off two conflicting goals: *1) Exploitation* suggests to use our current, but uncertain, approximation of the input-output relationship, based on the past query evaluations, to select the most promising input in terms of objective function value; and *2) Exploration* suggests to select the input that we are most uncertain about to improve our approximation of the input-output relationship.

The key ingredients of BO for data-driven decision making are: 1) *surrogate model* that captures our beliefs, based on past objective function evaluations, about the input-output relationship; and 2) *acquisition function* that scores each input according to the utility of querying the objective function on it next. The acquisition function uses the surrogate model of the true input-output relationship $f(x \in X)$ to decide which input to evaluate next by trading-off exploration and exploitation.

The surrogate model $\hat{f}(x)$ is a probabilistic model of the input-output relationship $f(x)$ trained on all input-output pairs from past objective function evaluations. It reflects our current beliefs about $f(x)$ and serves two purposes in BO. First, to guide exploitation, it allows us to cheaply estimate the

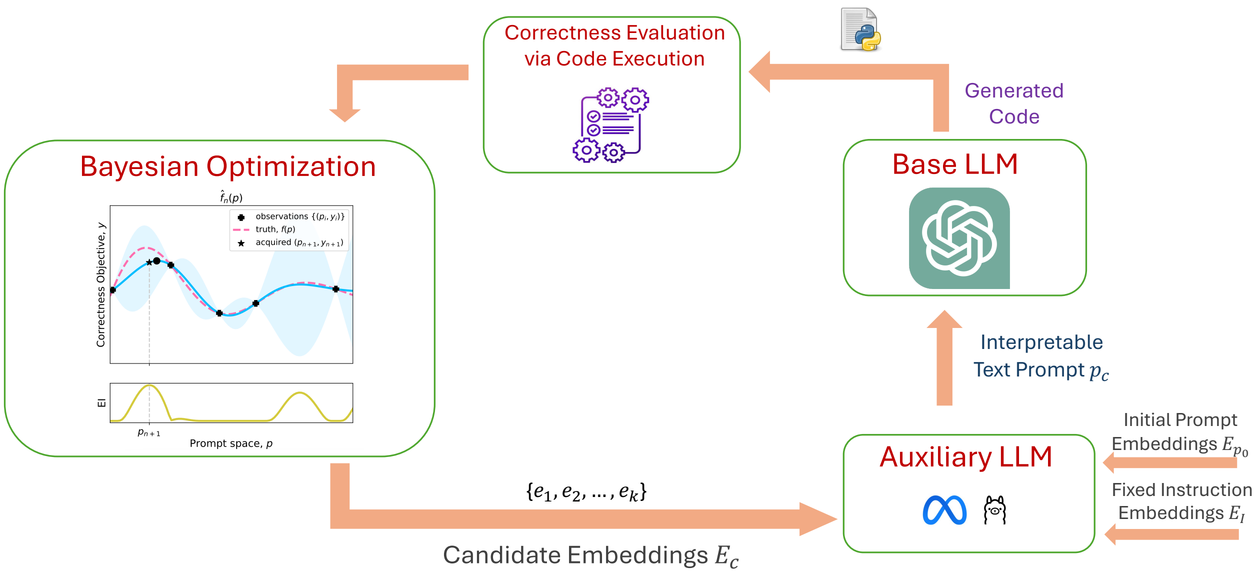

Correctness Evaluation via Code Execution
Generated Code
Bayesian Optimization
$\hat{f}_n(p)$
observations $\{(p_i, y_i)\}$
truth, $f(p)$
acquired (p_{n+1}, y_{n+1})
Correctness Objective, y
EI
p_{n+1}
Prompt space, p
Base LLM
Interpretable Text Prompt p_c
Auxiliary LLM
Initial Prompt Embeddings E_{p_0}
Fixed Instruction Embeddings E_I
$\{e_1, e_2, \ldots, e_k\}$
Candidate Embeddings E_c

Fig. 1: High-level overview of our BODE-GEN approach. The method begins with a set of candidate embeddings E_c=$\{e_1, e_2, \cdots, e_m\}$ proposed by the Bayesian Optimization algorithm. These embeddings combined with the initial prompt embeddings E_{p_0} and fixed instruction embeddings E_I are passed to the auxiliary LLM which generates an interpretable text prompt p_c. Subsequently, the base LLM is queried with this prompt p_c to generate the code which is evaluated for functional correctness through code execution on a set of developer provided test cases. The percentage of test cases passed by the code is used as the objective function value for the BO procedure. This overall procedure is repeated for a fixed number of iterations or until we find a prompt that generates code with 100 percent accuracy on the given test cases.

objective function value of all unevaluated inputs. Second, to guide exploration, variance quantifies the uncertainties in the predicted objective function value for the unevaluated inputs. This makes us aware of regions in input space we need to explore to improve our surrogate model and reduce the uncertainty in our beliefs about $f(x)$. Gaussian processes (GPs) [26] are the most commonly used surrogate models in BO owing to their flexibility as function approximators and principled uncertainty quantification.

The acquisition function scores the utility of evaluating the next input with the expensive objective function f. Here, "utility" is defined in terms of our ultimate goal of finding the optimal input with the minimum number of objective function evaluation queries. The acquisition function employs the prediction of the objective function value and the associated uncertainty from the surrogate model to assign a utility score to each candidate input that balances exploitation and exploration, respectively. The decision of which input to evaluate next is made by maximizing the acquisition function. Importantly, the acquisition function is cheap to evaluate. Some popular acquisition functions include expected improvement (EI) and upper confidence bound (UCB) [29].

To summarize, BO is an iterative procedure that is executed until we reach our goal or maximum iterations are reached. In each iteration, we select the input that maximizes the acquisition function for objective function evaluation and then update the surrogate model based on new training example.

B. BO-based Prompt Search via Continuous Embeddings

Algorithm 1 BODE-GEN Algorithm for Prompt Optimization

```
Require: Coding task $T$ and $n$ test cases; Initial prompt $p_0$;
    Base LLM $LLM_{base}$; and Auxiliary LLM $LLM_{aux}$
Ensure: Optimized prompt $\hat{p}$
 1: $E_{p_0} \leftarrow$ Embedding of initial prompt $p_0$ using $LLM_{aux}$
 2: $E_I \leftarrow$ Embedding of fixed instruction using $LLM_{aux}$
 3: Initialize surrogate model $\mathcal{M}$ with random projections and
    dimensionality-scaled priors [15] on a set of randomly
    initialized points.
 4: for iteration $t = 1$ to $T_{max}$ do
 5:     $E_c \leftarrow \arg\max_E$ AcquisitionFunction$(\mathcal{M}, E)$
 6:     $E_{comb} \leftarrow E_I \circ E_c \circ E_{p_0}$   ▷ Concatenated embedding
 7:     $p_c \leftarrow LLM_{aux}(E_{comb})$   ▷ Generate discrete prompt
 8:     $C \leftarrow LLM_{base}(p_c)$ ▷ Generate code using prompt $p_c$
 9:     Acc $\leftarrow$ Evaluate$(C, n)$ ▷ Accuracy on $n$ test cases
10:     Update surrogate model $\mathcal{M}$ with $(E_c, Acc)$
11: end for
12: return best found prompt $\hat{p}$ in terms of code accuracy
```

Much of the BO success is on continuous spaces with small number of input dimensions. There are two intertwined surrogate modeling challenges in applying BO for prompt search. First, as opposed to continuous inputs, modeling of combinatorial objects (e.g., sequences) is quite challenging because of a lack of general notion of smoothness on such objects. This is especially exacerbated in the *small-data regime* where we have access to only a small number of supervised examples from the input space. Second, the search space of prompts is high-dimensional. We provide principled solutions to address these challenges as part of our proposed BODE-GEN approach and explain their details below.

BODE-GEN performs search for optimized prompts in a continuous embedding space as opposed to the discrete prompt space. The key insight is to leverage an auxiliary open-source LLM LLM_{aux} (e.g., LLaMA 2) to bridge the gap between continuous embedding space and discrete prompt space. Specifically, we perform BO (both surrogate modeling and acquisition function optimization) in the continuous embedding space. The continuous search space for our BO method is parameterized as a set of d-dimensional embeddings $E=\{e_1, e_2, \cdots, e_m\}$, where each $e_i \in \mathbb{R}^d$ is a continuous vector in the d-dimensional embedding space of a local auxiliary LLM LLM_{aux}.

We perform the following sequence of steps in each iteration of BODE-GEN (see Algorithm 1 for pseudo-code and Figure 1 for illustration) given a surrogate model trained on the continuous embedding space.

1) Select the candidate input E_c from the continuous embedding space by maximizing the expected improvement (EI) acquisition function.
2) The selected embedding E_c is added as a suffix to the continuous embedding E_{p_0} of the initial prompt p_0. We also prepend the embeddings E_I for a simple instruction I: "`Your task is to rephrase/reformulate the code prompt given below to achieve a higher score on code generation by a large language model. Please provide the rephrased prompt in one block.`" given to the auxiliary LLM LLM_{aux} inorder to rephrase the original prompt p_0. The resulting combined embedding $E_{comb}=E_I \circ E_c \circ E_{p_0}$ where $\circ$ stands for concatenation operation.
3) The combined continuous embedding input E_{comb} is passed to the auxiliary LLM LLM_{aux} to generate a human-interpretable discrete prompt p_c.
4) The discrete prompt p_c is passed as input to the base LLM LLM_{base} to generate code C. The code C is executed on all n test cases to measure the functional accuracy, namely, ACCURACY(LLM_{base}, T).
5) If the functional accuracy of code C is 100 percent, we return code C as output. Otherwise, the surrogate model is updated using the new training example: input is the continuous embedding E_c and output is code accuracy.

Surrogate Modeling over High-Dimensions. Gaussian Process (GP) [26] based surrogate models are commonly used in real-world BO applications with small number of input dimensions (typically less than 50). However, the high-dimensional embedding space of auxiliary LLM poses a significant challenge for standard GP models. For example, the embedding dimension for LLama2 is 4096. GP models that are directly fitted on such a high-dimensional continuous space struggle to generalize, especially when the amount of available supervised data is limited, as is considered in this paper. We apply two synergistic techniques to tackle this challenge: random projections followed by dimensionality-scaled priors [15] for kernel hyper-parameters.

- **Random Projections**: We employ random projections [18] to reduce the dimensionality of our search space. The key intuition behind this approach is that, in high-dimensional spaces, most of the interesting structure in the data lies in a lower-dimensional manifold. Random projections can capture this structure while preserving important properties of the data, such as pairwise distances between points (as formalized by Johnson-Lindenstrauss lemma [17]). In the context of our prompt optimization setting, random projections allow us to work with a more manageable representation of the embedding space without significantly compromising the information content. Let $x \in \mathbb{R}^d$ be a point in our original high-dimensional embedding space, where d is large (e.g., 4096 for LLaMA 2). We aim to project this point onto a lower-dimensional space $\mathcal{Z} \in \mathbb{R}^k$, where $k \ll d$. The random projection is defined by a matrix $A \in \mathbb{R}^{k \times d}$, where each entry a_{ij} of matrix A is sampled independently from a standard normal distribution:
$$a_{ij} \sim \mathcal{N}(0, 1) \tag{2}$$
The projected point $z \in \mathcal{Z}$ is then obtained by:
$$z = Ax \tag{3}$$
After applying random projections, our GP surrogate model is defined on the low-dimensional space $\mathcal{Z}$.
- **Dimensionality scaled priors**: While random projections effectively reduce the dimensionality of our search space, the resulting projected space can still have hundreds of dimensions, posing challenges for standard GP models. GP surrogate models are entirely characterized by the choice of a kernel function (covariance function) $k(z, z')$ that measures the similarity between two input points z and z'. The choice of kernel function is critical as it encodes our prior beliefs about the function we are trying to model. Many canonical kernels such as RBF (Radial Basis Function) Kernel and Matern Kernel depend on a lengthscale augmented squared Euclidean distance $d(z, z')$ between z and z' i.e.
$$d(z, z') = \sum_{i}^{k} \left(\frac{z_i - z_i'^2}{l_i^2} \right) \tag{4}$$
The lengthscale $\{l_i\}$ is a critical hyper-parameter that captures the smoothness of functions represented by the kernel. In small supervised data settings (as in our problem), careful prior specification for this parameter is critical to achieve good performance on high-dimensional inputs. In order to address this challenge, we consider the recently proposed idea of scaling the prior on the

lengthscale hyper-parameters of the GP kernel with the square root of the input dimensionality of the search space [15]. Specifically, for a k-dimensional input space, dimensionality scaled prior for the length-scale parameter l_i for $i \in \{1, 2, \cdots, k\}$ is described as:

$$l_i \sim \text{LogNormal}(\mu = \log(\sqrt{k}), \sigma^2 = 1) \tag{5}$$

Overall, after applying random projections, our GP surrogate model is defined on the reduced space $\mathcal{Z}$ with dimensionality scaled priors on the lengthscale of the kernel $k(z, z')$.

IV. Experiments and Results

To evaluate the effectiveness of BODE-GEN, we investigate the following research questions (RQs):

- **RQ1**: *How effective is BODE-GEN for code generation?*
- **RQ2**: *How does BODE-GEN compare with CoT and OPRO prompting methods for code generation?*
- **RQ3**: *What changes are introduced by BODE-GEN to the intial prompt and how do they affect the resulting code?*

In what follows, we first describe our experimental setup including benchmark dataset, details of LLMs, configuration of BODE-GEN and baseline methods, and evaluation methodology. Next, we discuss the results to answer the three RQs.

A. Experimental Setup

Dataset. We benchmark our proposed BODE-GEN approach against strong baselines on the HumanEval+ benchmark [22] which is a recent extension of the widely-adopted HumanEval benchmark [8] for coding tasks. HumanEval consists of 164 python programming tasks, each containing a function signature and an initial prompt which is written as a docstring. The correctness of each task is measured by evaluating it on a set of pre-specified test cases. HumanEval+ extends the number of test cases in HumanEval tasks by 80x making it a challenging program synthesis benchmark.

Large Language Models. We employ three different LLM_{base} for code generation: one closed-source ChatGPT 3.5 (Turbo) [7], and two open source LLMs CodeLlama (7B) [28] and DeepSeekCoder [14]. Results with DeepSeekCoder are presented in the Appendix due to space. The instruction-tuned version of LLama2 (7B) [34] is used as the Auxiliary LLM LLM_{aux} for all the experiments.

Computing Machine. All our experiments were run on a machine featuring an Nvidia A40 GPU. The system is built on an x86_64 architecture with a 32-core AMD EPYC 7573X processor. The machine has 251 GB of memory.

Configuration of BODE-GEN. The search space for BODE-GEN is set to four embeddings, i.e., E=$\{e_1, e_2, \cdots, e_4\}$ where each $e_i \in \mathbb{R}^{4096}$. We randomly project each embedding to a 64 dimensional space resulting in a overall search space of 256 = (4 x 64) dimensions. We ran the BO method for T_{max}=50 iterations after initializing the surrogate model with 20 randomly picked embedding points (i.e., BO performance curves are shown for 70 iterations). Expected Improvement was picked as the acquisition function due to its practical success without any hyper-parameters and optimized over a discrete set of 10K points. We ran BO for multiple seeds (five) and present results with mean and error bars.

Baselines. We compare BODE-GEN with two strong baseline methods for code generation: Zero-shot Chain of Thought (CoT) [38] and Optimization by PROmpting (OPRO) [39]. CoT is a popular prompting approach that is shown to illicit reasoning from LLMs for complex tasks. Unlike the original formulation which requires providing multiple intermediate sequence of steps towards solving the problem, we consider the Zero-shot version where we append the key prompt "Let's think step by step" as a suffix to the initial prompt. OPRO is a state-of-the-art iterative prompt optimization technique that leverages LLMs to iteratively suggest better candidates conditioned on previously found prompts.

Evaluation Metric. For each coding task T, base language model LLM_{base} is queried with a candidate prompt to generate python code. We employ the `pass @1` metric defined below [8] to evaluate the correctness of the generated code:

$$\texttt{pass @1} = \mathbb{E}_{\text{tasks}}\left[1 - \binom{n-c}{1} / \binom{n}{1}\right] \tag{6}$$

where we generate $n = 3$ code samples per prompt to reduce the variance of the metric and c is the number of code samples that pass all unit tests for that task.

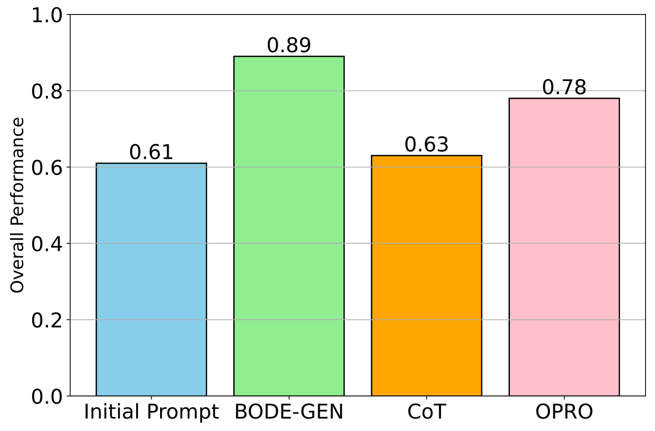
B. RQ1: Effectiveness of BODE-GEN

We measure the effectiveness of BODE-GEN in two ways: the code generation accuracy aggregated over all coding tasks in the HumanEval+ benchmark (higher the better) and the number of BO iterations to achieve high code generation accuracy (smaller the better). BODE-GEN demonstrates significant improvements in finding prompts that improve the generated code's correctness accuracy compared to the baseline methods. The results, as depicted in Figure 2a, show that BODE-GEN achieves an average code generation accuracy of 0.89, which is notably higher than that of initial prompts (0.61), CoT (0.63), and OPRO (0.78). Figure 2b, shows that BODE-GEN archives an average code generation accuracy of 0.2, which is notably higher than that of initial prompts (0.23), CoT (0.22), and OPRO (0.38). This substantial increase in performance (code accuracy) highlights the efficacy of BODE-GEN in finding prompts that generate functionally correct code on the HumanEval+ benchmark.

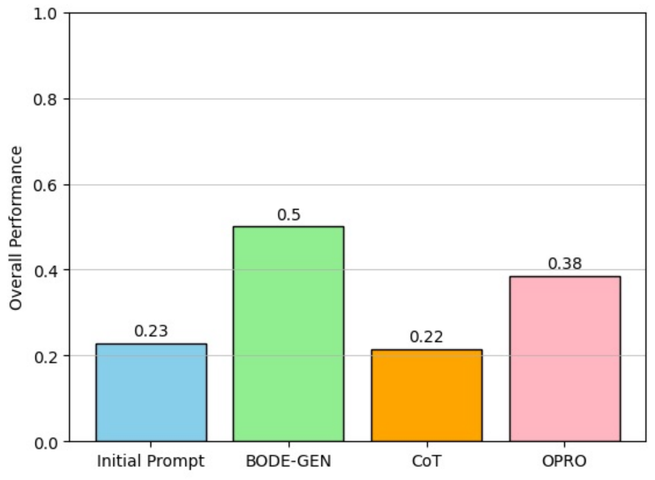
We also show the progress of BODE-GEN in terms of the generated code's accuracy as a function of number of BO iterations (i.e., number of tried prompts) in Figure 3, 5, 4 on some representative coding tasks noting that our findings are similar on other coding tasks. In all cases, BODE-GEN demonstrates rapid improvement in the objective value (percentage of test cases passed) within the first 20-30 BO iterations, often achieving near-optimal performance by the 50th iteration. This rapid convergence suggests that BODE-GEN efficiently explores the prompt space to find optimized prompts for code generation. As evident from the figures, `BODE-GEN` iteratively finds better prompts that are able to reach 100% correctness accuracy.

C. RQ2: Comparison to Baselines CoT and OPRO

In comparison to CoT and OPRO, `BODE-GEN` outperforms both methods significantly as shown in Figure 2a, and 2b. CoT shows only small improvement over the initial prompt and does significantly worse than `BODE-GEN`. OPRO achieves better results compared to CoT since it is an iterative approach (similar to `BODE-GEN`) that finds better prompts iteratively. We define a notion of task difficulty as the correctness of the generated code achieved by initial fixed prompt given for each task and create three groups (easy, medium, and hard) of increasing difficult to evaluate all three methods. The results are shown in Figure 6, 7 which shows that the gap between `BODE-GEN` and the performance of baselines increases as we increase the difficulty of the task. This means that `BODE-GEN` is more effective than baselines for hard coding tasks.



(a) Results for Base LLM: ChatGPT 3.5(Turbo)



(b) Results for Base LLM: CodeLlama-7b

Fig. 2: Results comparing the overall performance of `BODE-GEN` and baselines with (a) ChatGPT 3.5(Turbo) and (b) CodeLlama-7b as the base LLM. Here, overall performance is computed as the percentage of tasks (out of total 164) from HumanEval+ solved to 100% correctness.

D. RQ3: Qualitative Analysis

To understand how our BO-based prompt optimization approach mproves LLM-driven program synthesis in terms of the correctness of the resulting code, we conducted qualitative analysis on 15 randomly selected coding task cases among those in which the correctness improvement was the most substantial (i.e., the most challenging cases for the LLMs with the *original prompts* used in the HumanEval+ dataset [21]). By examining these cases, we aim to (1) identify the key changes in the prompt that our optimization approach makes, and (2) based on these changes and how the generated code differs between the initial prompt and optimized prompt, distill common patterns of and main insights into what makes a prompt effective leading LLMs to produce correct code.

1) Key Changes Induced in Prompt Optimizations: For each of these chosen cases, we carefully compare the two versions of the code-generation prompt, aiming to identify key differences between them in all possible aspects that may affect the base LLM's ability to generate correct code.

Change 1: Use of Examples. While providing (e.g., input/output) examples generally help LLMs generate correct code, how the examples are used in the prompt matter. The original prompts typically include examples within the `docstring`, which might be less visible. In optimized prompts from BO, the examples are clearly separated from the main instructions, making them more prominent and easier to reference. For example, in the case of the `Multiply` function, shown in Figure 8, the original prompt provides examples within the `docstring`, versus our optimized prompt listing the examples in plain text after describing the task. By clearly separating instructions from examples rather than combining them in one information block, these changes improve LLMs' code correctness through *separation of concerns*.

Change 2: Instruction Clarity and Detail. In several studied cases, the original prompt often includes instructions within the code's `docstring`, which might be concise but less explicit. In the optimized version, the prompt provides a detailed, narrative description of the task, often in plain English, making the instructions clearer and more explicit. For example, in the case of `max_fill` function generation, as shown in Figure 9, the original prompt (left) describes the task as "You are given a rectangular grid of wells...", while the improved prompt provides more details and clarity with "Please write a function called max_fill that takes in a rectangular grid of wells and a capacity as input...".

Change 3: Structure and Organization. In most of the studied cases, the original prompts have a less organized structure, while the optimization-resulted prompts use a more structured, step-by-step format, often with clear separations between different parts of the instructions. As seen in Figure 10, for generating the `get_closest_vowel` function, our optimized prompt improves how the entire prompt is

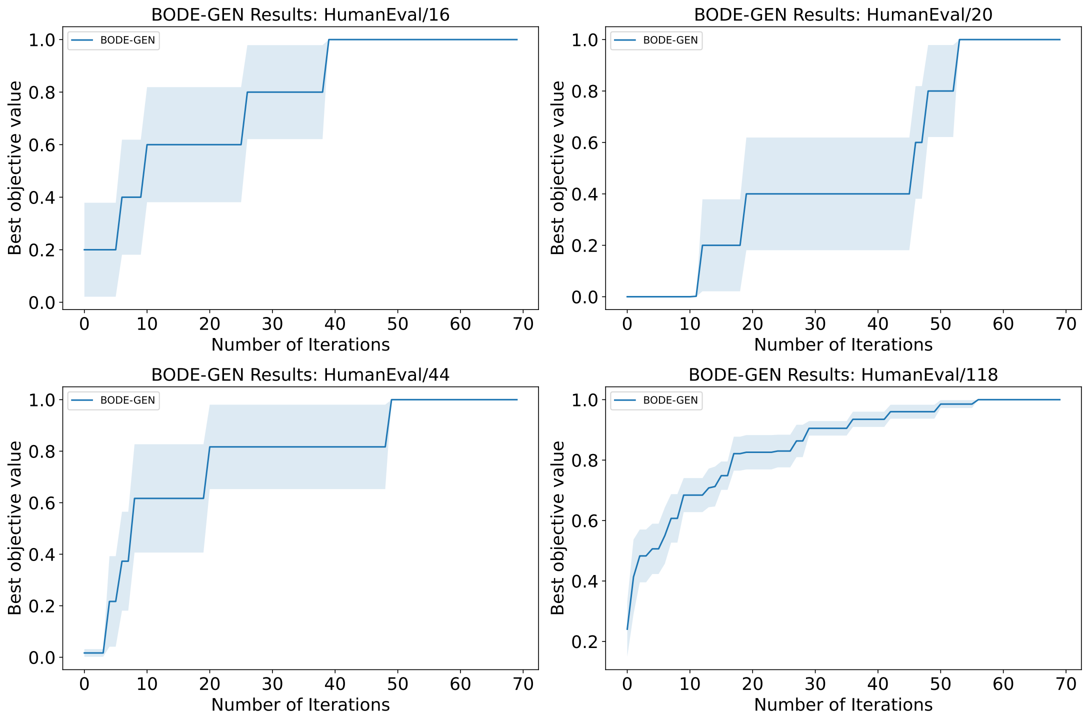


Fig. 3: Results showing BODE-GEN's performance on ChatGPT 3.5(Turbo) as base-LLM as a function of number of iterations (number of base LLM calls with different prompts) on a subset of representative tasks from HumanEval+ benchmark. Note that the objective value for BO is the percentage of test cases passed by the generated code for a given coding task. As shown in the figure, prompts suggested by BODE-GEN are consistently able to reach 100% code generation correctness. Each BO iteration corresponds to roughly one query to the base LLM (precisely it is three queries per iteration since we generate three samples for each prompt to compute `pass @1`).

structured: both the assumptions and examples are clearly organized in addition to the task description.

Change 4: Language and Readability. Another main prompt change induced by our optimization lies in the language use in the prompt that affects its readability. In particular, the original prompts tend to use a technical and formal style typical of in-code documentation, while the optimized prompts use plain English and a more narrative style, improving readability and comprehension. Take the `max_fill` function (Figure 9) as an example again, the improved version of the prompt uses a narrative and explanatory style, versus the more technical style of the original prompt. Intuitively, LLMs are trained on more natural language corpus than technical documents, which may justify why these changes that improve readability help produce LLMs produce more correct code.

Change 5: Descriptive Guidance. The original prompts often provide guidance on what the function should do in a general manner. In contrast, the optimized prompt has more specific guidance on how to approach the task, including iterative processes or specific logic to follow. As shown in Figure 9, if we put aside the detailed examples, the original task description itself is overall general. In the optimized version, much more specific guidance is included on iterating through the grid and tracking bucket usage, which helped the LLMs correct the errors in the code generated with the original prompt, even without using those examples.

To summarize, the common differences between the two versions of each prompt in our studied cases highlight the importance of clarity, detail, explicitness, and structure in writing prompts for code generation. By ensuring that *instructions are clear, detailed, and well-organized*, and by *providing explicit guidance and examples*, the quality and accuracy of the generated code can be significantly improved.

2) Patterns of Correct-Code-Generating Prompts: Based on these prompt changes made by our optimization approach as summarized above, together with comparing the code generated by the two prompt versions, we further identified the following major patterns of correct-code-generating prompts.

Pattern 1: Clear and Detailed Instructions. We observed that providing detailed and explicit instructions helps LLMs understand the requirements better, leading to more accurate code generation. In the case of `max_fill` function (Figure 9), for example, the original prompt states "Your task is to use the buckets to empty the wells.", which is much less elaborate and explicit than the optimized version: "The function should solve the problem by iterating through the rows of the grid, keeping track of the number of times each bucket needs to be lowered to empty each well, and returning the total number

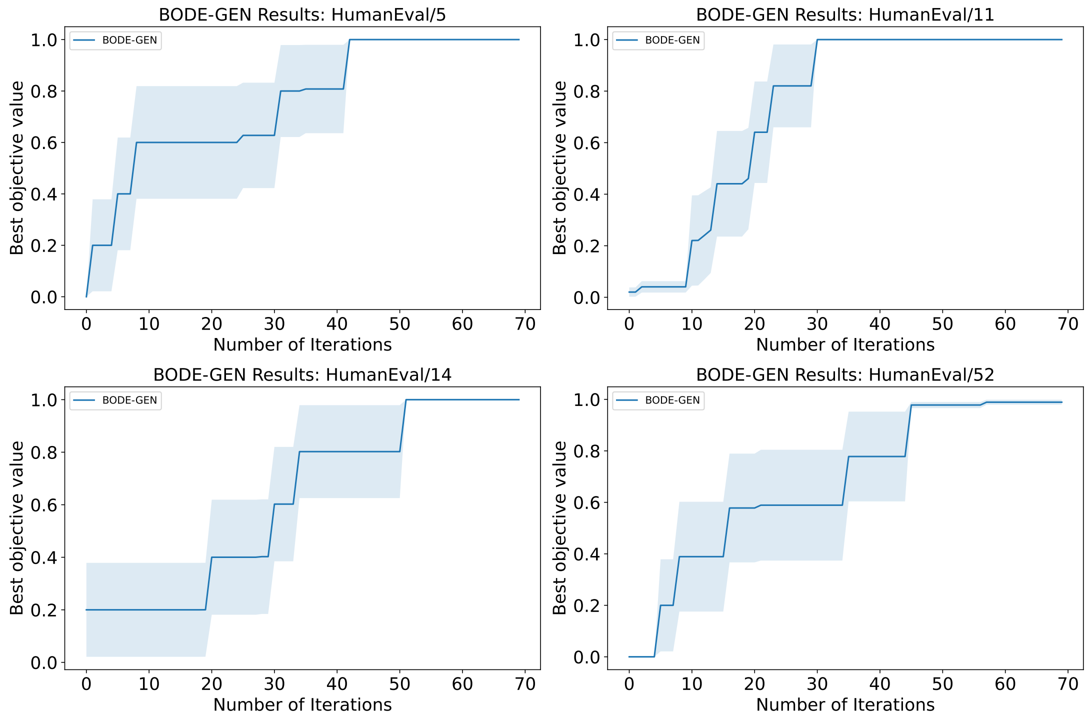


Fig. 4: Results showing BODE-GEN's performance on CodeLlama-7b as base-LLM as a function of number of iterations (number of base LLM calls with different prompts) on a subset of representative tasks from HumanEval+ benchmark.

of bucket lowerings needed."

The *detailed instructions in the optimized prompt clarify the method to solve the problem, resulting in better code generation.*

Pattern 2: Step-by-Step Breakdown. From multiple cases, it appears clear that breaking down the task into smaller, clear steps helps the model generate code that follows the intended logic more closely. One example is found in the prompt for synthesizing the `numerical_letter_grade` function, a shown in Figure 11. The original version simply provides a context and a list of GPAs, offering no intermediate reasoning. The optimized version of the prompt clearly describes the function's purpose, parameters, expected behavior, and gives a table for GPA to letter grade mapping. The *step-by-step breakdown ensures that the model accurately follows the logic needed* to map GPAs to letter grades.

Pattern 3: Explicit Constraints and Assumptions. Inspection of the optimized prompts reveals that specifying constraints and assumptions ensures LLMs adhere to the necessary conditions and edge cases. As illustrated in Figure 10, the original prompt does provide assumptions and examples but not explicitly (i.e., within a `docstring`), while the optimized version clearly lists the constraints and examples in plain text (i.e., explicitly). This shows that *explicitly mentioning constraints and assumptions helps the model generate code that respects those conditions.*

Pattern 4: Separation of Examples and Instructions. Almost all of our studied cases confirm that including (e.g., input/output) examples in code-generating prompts help LLMs produce correct code, but it is essential to separate examples from instructions. The separation ensures clarity and helps the model focus on understanding both the requirements and the examples. As shown in Figure 10, the optimized prompt, which lists examples separately after the task description, improves over the original prompt which combines examples and the task description (within the `docstring`). Apparently, *clearly separating examples from instructions enhances LLMs' ability to parse and understand both parts effectively.*

Pattern 5: Plain English Descriptions. As evidenced in many cases, using plain English to describe the task makes it easier for LLMs to parse the requirements and generate correct code. As seen in Figure 9, the original prompt uses a technical and formal style, while the improved prompt provides a narrative and explanatory style. *Plain English task descriptions make it easier for the model to understand the task and generate correct logic.*

In summary, to write better prompts for code generation with LLMs, it is crucial to *provide clear, detailed instructions, break down tasks into smaller steps, specify constraints and assumptions explicitly, use multiple examples, describe tasks in plain English, and separate examples from instructions*. These patterns help the model understand the requirements better and generate more accurate and correct code.

E. Additional Results on CodeLlama

In this section, we present additional results comparing BODE-GEN with the baseline approaches on another base

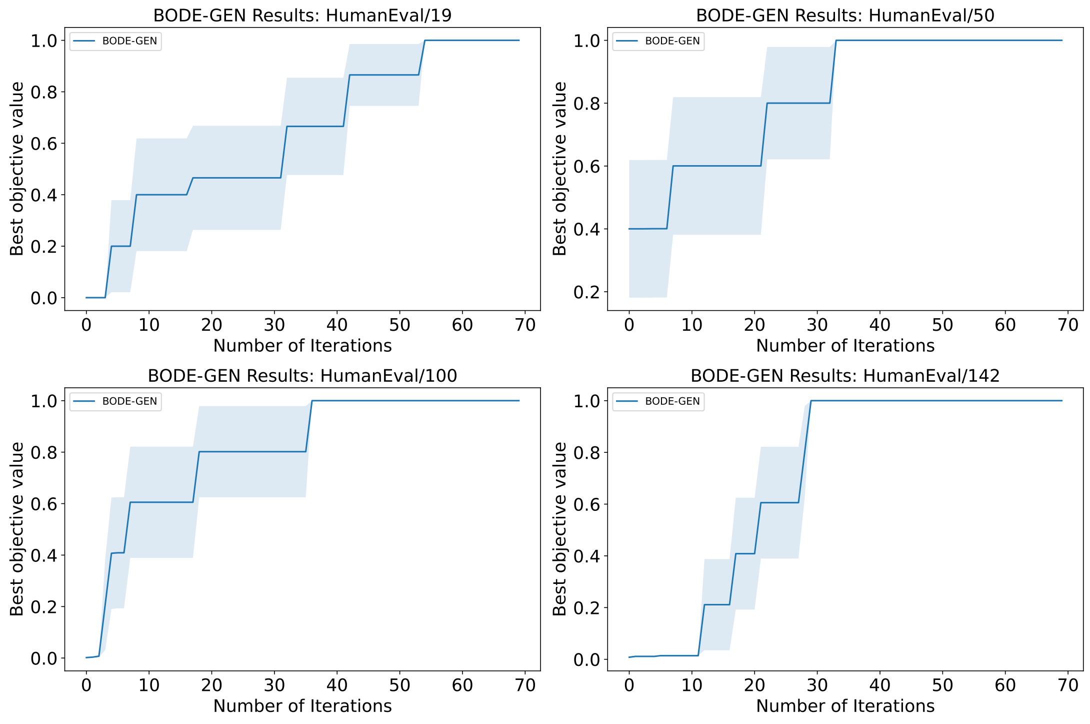


Fig. 5: Results showing BODE−GEN's performance on DeepSeek-Coder as base-LLM as a function of number of iterations (number of base LLM calls with different prompts) on a subset of representative tasks from HumanEval+ benchmark.

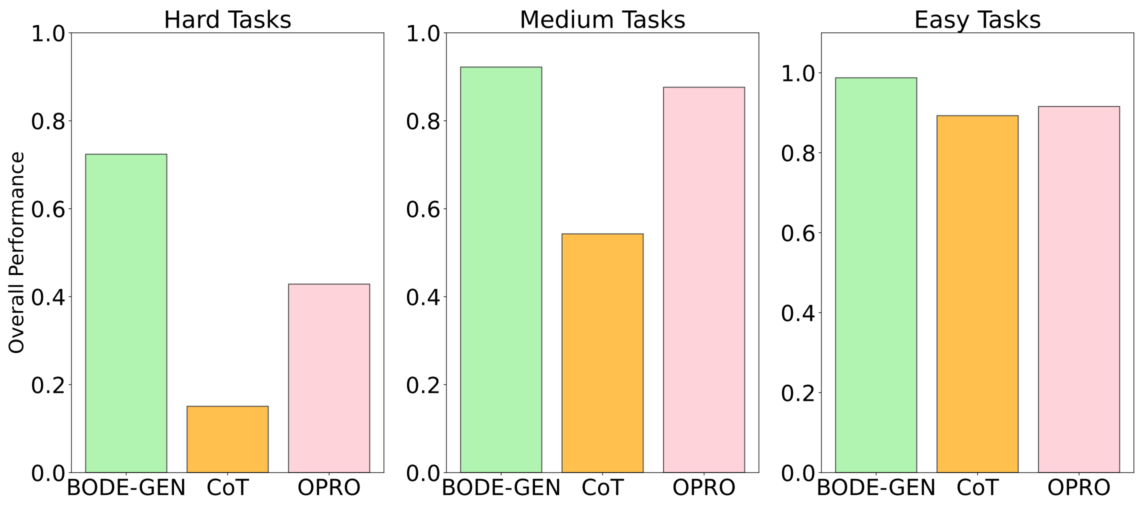


Fig. 6: Results comparing the overall performance of BODE−GEN with zero-shot CoT and OPRO with ChatGPT 3.5(Turbo) as base-LLM on a grouping of HumanEval+ tasks based on a notion of difficulty measured as the correctness of the code generated by initial prompts given for each task. For example, the easy/hard class refers to all tasks for which the code generated via initial prompt achieves correctness (above 67%/below 30%) respectively. The medium class contains all tasks with their initial prompts' correctness between 30-67%.

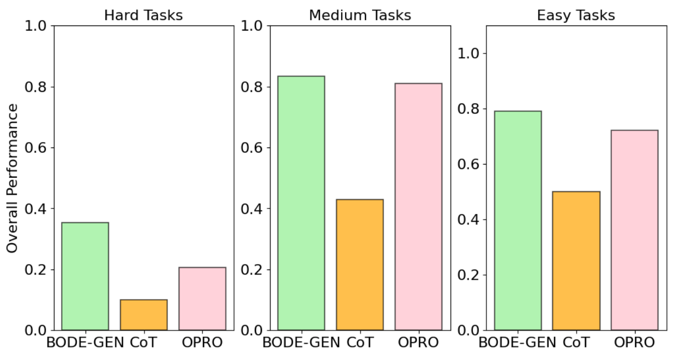
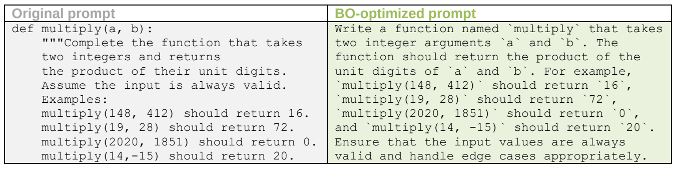


Fig. 7: Results of BODE-GEN and baselines with CodeLlama-7b as base-LLM on a grouping of HumanEval+ tasks based on a notion of difficulty measured as the correctness of the code generated by initial prompts given for each task. For example, the easy/hard class refers to all tasks for which the code generated via initial prompt achieves correctness (above 67%/below 30%) respectively. The medium class contains all tasks with their initial prompts' correctness between 30-67%.



Original prompt

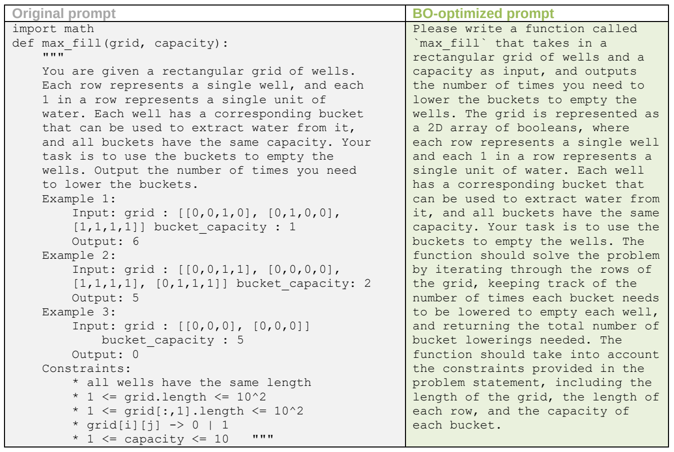
```
def multiply(a, b):
    """Complete the function that takes
    two integers and returns
    the product of their unit digits.
    Assume the input is always valid.
    Examples:
    multiply(148, 412) should return 16.
    multiply(19, 28) should return 72.
    multiply(2020, 1851) should return 0.
    multiply(14,-15) should return 20.
```

BO-optimized prompt

```
Write a function named `multiply` that takes
two integer arguments `a` and `b`. The
function should return the product of the
unit digits of `a` and `b`. For example,
`multiply(148, 412)` should return `16`,
`multiply(19, 28)` should return `72`,
`multiply(2020, 1851)` should return `0`,
and `multiply(14, -15)` should return `20`.
Ensure that the input values are always
valid and handle edge cases appropriately.
```

Fig. 8: Original prompt versus BO-optimized prompt for generating the `multiply` function (HumanEval+ Case 97).

LLM, namely CodeLlama. Due to limited computing resource availability, we picked a subset of randomly picked 50 tasks from HumanEval+ for generating these results. As shown in Figure **??**, BODE-GEN outperforms all the baselines in finding prompts that generate code with higher correctness and the gap between the baselines and BODE-GEN increases on difficult tasks as shown in Figure **??**. The absolute performance



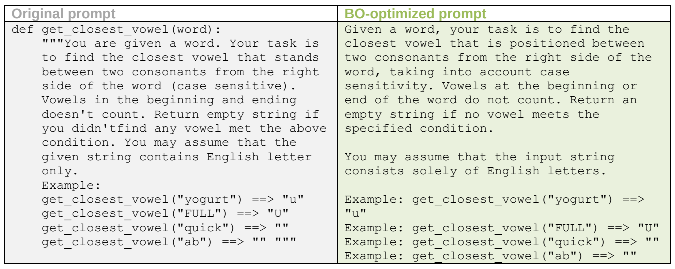
Original prompt

```
import math
def max_fill(grid, capacity):
    """
    You are given a rectangular grid of wells.
    Each row represents a single well, and each
    1 in a row represents a single unit of
    water. Each well has a corresponding bucket
    that can be used to extract water from it,
    and all buckets have the same capacity. Your
    task is to use the buckets to empty the
    wells. Output the number of times you need
    to lower the buckets.
    Example 1:
        Input: grid : [[0,0,1,0], [0,1,0,0],
        [1,1,1,1]] bucket_capacity : 1
        Output: 6
    Example 2:
        Input: grid : [[0,0,1,1], [0,0,0,0],
        [1,1,1,1], [0,1,1,1]] bucket_capacity: 2
        Output: 5
    Example 3:
        Input: grid : [[0,0,0], [0,0,0]]
            bucket_capacity : 5
        Output: 0
    Constraints:
        * all wells have the same length
        * 1 <= grid.length <= 10^2
        * 1 <= grid[:,1].length <= 10^2
        * grid[i][j] -> 0 | 1
        * 1 <= capacity <= 10   """
```

BO-optimized prompt

```
Please write a function called
`max_fill` that takes in a
rectangular grid of wells and a
capacity as input, and outputs
the number of times you need to
lower the buckets to empty the
wells. The grid is represented as
a 2D array of booleans, where
each row represents a single well
and each 1 in a row represents a
single unit of water. Each well
has a corresponding bucket that
can be used to extract water from
it, and all buckets have the same
capacity. Your task is to use the
buckets to empty the wells. The
function should solve the problem
by iterating through the rows of
the grid, keeping track of the
number of times each bucket needs
to be lowered to empty each well,
and returning the total number of
bucket lowerings needed. The
function should take into account
the constraints provided in the
problem statement, including the
length of the grid, the length of
each row, and the capacity of
each bucket.
```

Fig. 9: Original prompt versus BO-optimized prompt for generating the `max_fill` function (HumanEval+ Case 115).



Original prompt

```
def get_closest_vowel(word):
    """You are given a word. Your task is
    to find the closest vowel that stands
    between two consonants from the right
    side of the word (case sensitive).
    Vowels in the beginning and ending
    doesn't count. Return empty string if
    you didn'tfind any vowel met the above
    condition. You may assume that the
    given string contains English letter
    only.
    Example:
    get_closest_vowel("yogurt") ==> "u"
    get_closest_vowel("FULL") ==> "U"
    get_closest_vowel("quick") ==> ""
    get_closest_vowel("ab") ==> "" """
```

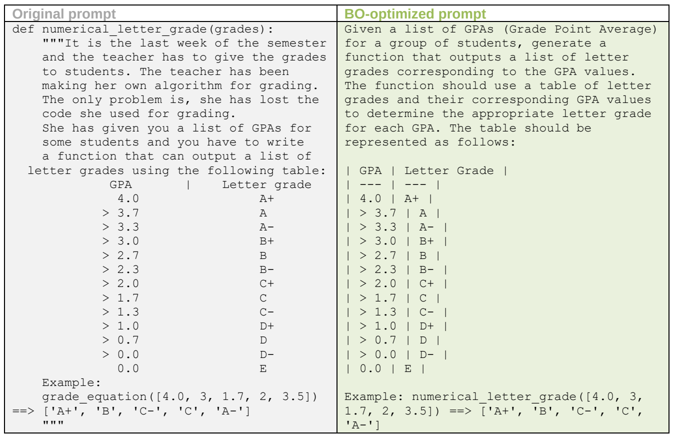
BO-optimized prompt

```
Given a word, your task is to find the
closest vowel that is positioned between
two consonants from the right side of the
word, taking into account case
sensitivity. Vowels at the beginning or
end of the word do not count. Return an
empty string if no vowel meets the
specified condition.

You may assume that the input string
consists solely of English letters.

Example: get_closest_vowel("yogurt") ==>
"u"
Example: get_closest_vowel("FULL") ==> "U"
Example: get_closest_vowel("quick") ==> ""
Example: get_closest_vowel("ab") ==> ""
```

Fig. 10: Original prompt versus BO-optimized prompt for generating the `get_closest_vowel` function (HumanEval+ Case 118).



Original prompt

```
def numerical_letter_grade(grades):
    """It is the last week of the semester
    and the teacher has to give the grades
    to students. The teacher has been
    making her own algorithm for grading.
    The only problem is, she has lost the
    code she used for grading.
    She has given you a list of GPAs for
    some students and you have to write
    a function that can output a list of
  letter grades using the following table:
           GPA       |    Letter grade
            4.0                A+
          > 3.7                A
          > 3.3                A-
          > 3.0                B+
          > 2.7                B
          > 2.3                B-
          > 2.0                C+
          > 1.7                C
          > 1.3                C-
          > 1.0                D+
          > 0.7                D
          > 0.0                D-
            0.0                E
    Example:
    grade_equation([4.0, 3, 1.7, 2, 3.5])
==> ['A+', 'B', 'C-', 'C', 'A-']
    """
```

BO-optimized prompt

```
Given a list of GPAs (Grade Point Average)
for a group of students, generate a
function that outputs a list of letter
grades corresponding to the GPA values.
The function should use a table of letter
grades and their corresponding GPA values
to determine the appropriate letter grade
for each GPA. The table should be
represented as follows:

| GPA | Letter Grade |
| --- | --- |
| 4.0 | A+ |
| > 3.7 | A |
| > 3.3 | A- |
| > 3.0 | B+ |
| > 2.7 | B |
| > 2.3 | B- |
| > 2.0 | C+ |
| > 1.7 | C |
| > 1.3 | C- |
| > 1.0 | D+ |
| > 0.7 | D |
| > 0.0 | D- |
| 0.0 | E |

Example: numerical_letter_grade([4.0, 3,
1.7, 2, 3.5]) ==> ['A+', 'B', 'C-', 'C',
'A-']
```

Fig. 11: Original prompt versus BO-optimized prompt for generating the `numerical_letter_grade` function (HumanEval+ Case 81).

of all methods including initial Prompt, CoT, OPRO, and BODE-GEN is relatively low using CodeLlama as the base LLM when compared to the corresponding results when ChatGPT 3.5(Turbo) is used as the base LLM. The main reason for this behavior is that ChatGPT is a more powerful LLM than CodeLlama. As can be seen from the results in Figure **??**, BODE-GEN still significantly improves the correctness of generated code over baseline methods.

V. Threats to Validity

The main internal validity threat lies in possible implementation errors in our tool and experimental scripts. To minimize this threat, we have performed careful code reviews of these implementations against manageable testing scenarios. Another major issue LLMs are commonly subject to are hallucination [33], which in our study may cause inconsistent and unreliable prompt improvements. To deal with such issues, we set the temperature of the auxiliary LLM to zero in our experiments and compute correctness accuracy (`pass @1`) by generating multiple samples from the base LLM. We also ran each experiment multiple times and took results that are consistent among the runs. Nevertheless, this mitigation may not have entirely ruled out the possibility that developers may not always get the same correctness improvement when using our technique for prompt optimization, or they may need slightly more iterations to get the same improvement as shown in our evaluation.

The main external validity threat concerns the datasets we used, as well as LLMs and baseline approaches chosen, in our experiments. We used a benchmark (HumanEval+) that is popularly used in LLM evaluations. Yet the prompts in it may not represent real-world code-synthesis prompts developers use. Meanwhile, we chose capable LLMs that are affordable to use with respect to the scale of our experiments. With the rapid evolution of LLMs, more advanced models may not guaranteed to be improved as much as what we present. Similarly, we selected CoT as one of our baselines, which is a state-of-the-art prompting strategy on LLMs. Even more advanced prompting could have achieved better performance (although how to instantiate them for code generation remains an open problem as of now).

VI. Related Work

We provided an overview of the broader literature on LLMs for code generation in the introduction section. Below we discuss closely related work to our specific problem setting.

LLM-driven Code Generation. To address the challenge that LLMs often generate incorrect code [3], [21], prior work has explored various strategies. One category includes knowledge augmentation [2], [16] and reasoning elicitation [1], [23]. Chain-of-thoughts (CoT) prompting [38] has been shown to be useful to improve LLM-driven code generation [23]. However, our experiments on HumanEval benchmark show that the improvement with CoT is small. Jigsaw [16] performs post-processing of the generated code to check and calibrate correctness, by augmenting the LLMs with knowledge about the syntax and semantics of programs based on program analysis and synthesis techniques. This approach is not automatic and maybe incomplete for some coding tasks. Ahmed et al., [1] leverage the self-consistency technique, which has previously been shown promising for improving the reasoning capabilities of LLMs via CoT, for generating bug fixes by leveraging associated commit logs as explanations. However, it is not clear how to apply this method for synthesizing programs from scratch. Recent work by Murr et al., [24] found that the specificity of prompts has a major impact on the quality of LLM-generated code: more specific prompts tend to produce accurate code meeting the functional requirements although compromising the generation diversity. However, determining the specificity level is currently a manual process.

A recent prompting technique, AceCoder [19], is proposed particularly for code generation with LLMs. It starts with asking the LLM to analyze the given requirements and output an intermediate preliminary output (e.g., test cases), which is then utilized to retrieve similar programs that meet the requirements as exemplars in the code-generation prompt. However, such code examples may not always be available for a given arbitrary requirement. Following a different strategy, AlphaCodium [27] is proposed to leverage test-based iterative refinement of the code generation process based on LLMs, where the tests are from what is already available and generated by the AI models. The effectiveness of AlphaCodium relies on the quality and availability of such tests, which itself is an unresolved challenge. Tao et al. [32] employs grammar-guided evolutionary search to improve LLM-based program synthesis. It uses the LLM-generated code as an initial input for the evolutionary search process after a grammar-mapping phase, which allows for program development and fixing errors. Subsequently, it employs different similarity metrics related to the LLM-generated code to steer the multi-objective evolutionary search process. However, the evaluation is only limited to grammar validation of the generated code as opposed to functional correctness.

Optimization by PROmpting (OPRO) is the state-of-the-art technique for optimizing tasks specified in natural language. OPRO uses LLMs to generate new solutions based on previously evaluated solutions and their scores, iteratively improving the objective function. Our experiments on HumanEval benchmark demonstrates that OPRO is better than CoT but it is less effective than our proposed BODE-GEN approach in terms of code generation accuracy.

In contrast to prior work, our proposed BODE-GEN approach for optimizing prompts is *fully automatic* (no manual intervention) and *sample-efficient* (minimized the number of prompts tried) by leveraging the test cases for coding task as per the test-driven software development paradigm [5].

Bayesian Optimization. BO has shown a lot of success in optimizing continuous spaces with small number of dimensions (typically less than 50). There is relatively less work on BO over discrete spaces which is significantly challenging than BO over continuous spaces [10] and is limited to a small number of dimensions. We solve the problem of BO over discrete prompt space by reducing it to BO over high-dimensional continuous embedding space using an auxiliary LLM. We proposed novel algorithmic solutions to build effective surrogate models in the small training data setting to handle challenges of the high-dimensional continuous embedding space.

VII. Summary and Future Work

This paper developed and studied `BODE-GEN`, a novel Bayesian optimization approach for prompt search in large language models (LLMs) for code generation tasks. We formulate this search as optimization over high-dimensional continuous embedding space of an auxiliary LLM. By leveraging random projections and dimensionality-scaled priors, our method effectively handles the challenges of high dimensionality in the embedding space. Extensive experiments on the HumanEval+ benchmark demonstrate `BODE-GEN`'s effectiveness to significantly improve code generation accuracy across a wide variety of coding tasks when compared to strong baselines. As LLMs continue to play a crucial role in software development, `BODE-GEN` offers a new automated tool for improving their reliability and effectiveness, ultimately improving developer's productivity. Future work includes reducing the dependency of `BODE-GEN` on the requirement of test cases for coding tasks.

References

[1] T. Ahmed and P. Devanbu. Better patching using llm prompting, via self-consistency. In *2023 38th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 1742–1746. IEEE, 2023.

[2] T. Ahmed, K. S. Pai, P. Devanbu, and E. Barr. Automatic semantic augmentation of language model prompts (for code summarization). In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[3] J. Austin, A. Odena, M. Nye, M. Bosma, H. Michalewski, D. Dohan, E. Jiang, C. Cai, M. Terry, Q. Le, et al. Program synthesis with large language models. *arXiv preprint arXiv:2108.07732*, 2021.

[4] A. Balayn, M. Yurrita, F. Rancourt, F. Casati, and U. Gadiraju. An empirical exploration of trust dynamics in llm supply chains. *arXiv preprint arXiv:2405.16310*, 2024.

[5] K. Beck. *Test-Driven Development by Example*. Addison Wesley. ISBN 978-0-321-14653-3, 2002.

[6] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[7] T. B. Brown, B. Mann, N. Ryder, M. Subbiah, J. D. Kaplan, P. Dhariwal, A. Neelakantan, P. Shyam, G. Sastry, A. Askell, et al. Language models are few-shot learners. In *Advances in Neural Information Processing Systems*, pages 1877–1901, 2020.

[8] M. Chen, J. Tworek, H. Jun, Q. Yuan, H. P. de Oliveira Pinto, J. Kaplan, H. Edwards, Y. Burda, N. Joseph, G. Brockman, A. Ray, N. Puri, G. Krueger, M. Petrov, H. Khlaaf, G. Sastry, P. Mishkin, B. Chan, S. Gray, N. Ryder, M. Pavlov, A. Power, L. Kaiser, M. Bavarian, C. Winter, P. Tillet, F. P. Such, D. Cummings, M. Plappert, F. Chantzis, E. Barnes, A. Herbert-Voss, W. H. Guss, A. Nichol, A. Paino, N. Tezak, J. Tang, I. Babuschkin, S. Balaji, S. Jain, A. Saunders, B. Houghton, J. Pfau, D. de Las Casas, L. Bottou, C. Choi, A. Coates, S. McCandlish, A. Radford, I. Sutskever, and D. Amodei. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

[9] T. Coignion, C. Quinton, and R. Rouvoy. A performance study of llm-generated code on leetcode. In *Proceedings of the 28th International Conference on Evaluation and Assessment in Software Engineering*, pages 79–89, 2024.

[10] A. Deshwal, S. Belakaria, and J. R. Doppa. Mercer features for efficient combinatorial bayesian optimization. In *Proceedings of the AAAI Conference on Artificial Intelligence*, volume 35, pages 7210–7218, 2021.

[11] R. J. Ellison, J. B. Goodenough, C. B. Weinstock, and C. Woody. Evaluating and mitigating software supply chain security risks. *Software Engineering Institute, Tech. Rep. CMU/SEI-2010-TN-016*, 2010.

[12] W. Enck and L. Williams. Top five challenges in software supply chain security: Observations from 30 industry and government organizations. *IEEE Security & Privacy*, 20(2):96–100, 2022.

[13] T. Gao, A. Fisch, and D. Chen. Making pre-trained language models better few-shot learners. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics (ACL)*, pages 3816–3830, 2021.

[14] D. Guo, Q. Zhu, D. Yang, Z. Xie, K. Dong, W. Zhang, G. Chen, X. Bi, Y. Wu, Y. Li, et al. Deepseek-coder: When the large language model meets programming–the rise of code intelligence. *arXiv preprint arXiv:2401.14196*, 2024.

[15] C. Hvarfner, E. O. Hellsten, and L. Nardi. Vanilla bayesian optimization performs great in high dimensions. In *Forty-first International Conference on Machine Learning*, 2024.

[16] N. Jain, S. Vaidyanath, A. Iyer, N. Natarajan, S. Parthasarathy, S. Rajamani, and R. Sharma. Jigsaw: Large language models meet program synthesis. In *Proceedings of the 44th International Conference on Software Engineering*, pages 1219–1231, 2022.

[17] K. G. Larsen and J. Nelson. Optimality of the johnson-lindenstrauss lemma. In *2017 IEEE 58th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 633–638. IEEE, 2017.

[18] B. Letham, R. Calandra, A. Rai, and E. Bakshy. Re-examining linear embeddings for high-dimensional bayesian optimization. *Advances in neural information processing systems*, 33:1546–1558, 2020.

[19] J. Li, Y. Zhao, Y. Li, G. Li, and Z. Jin. Acecoder: An effective prompting technique specialized in code generation. *ACM Transactions on Software Engineering and Methodology*, 2024.

[20] F. Liu, Y. Liu, L. Shi, H. Huang, R. Wang, Z. Yang, and L. Zhang. Exploring and evaluating hallucinations in llm-powered code generation. *arXiv preprint arXiv:2404.00971*, 2024.

[21] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[22] J. Liu, C. S. Xia, Y. Wang, and L. Zhang. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *Advances in Neural Information Processing Systems*, 36, 2024.

[23] Y. Ma, Y. Yu, S. Li, Y. Jiang, Y. Guo, Y. Zhang, Y. Xie, and X. Liao. Bridging code semantic and llms: Semantic chain-of-thought prompting for code generation. *arXiv preprint arXiv:2310.10698*, 2023.

[24] L. Murr, M. Grainger, and D. Gao. Testing llms on code generation with varying levels of prompt specificity. *arXiv preprint arXiv:2311.07599*, 2023.

[25] D. Nam, A. Macvean, V. Hellendoorn, B. Vasilescu, and B. Myers. Using an llm to help with code understanding. In *Proceedings of the IEEE/ACM 46th International Conference on Software Engineering*, pages 1–13, 2024.

[26] C. E. Rasmussen and C. K. I. Williams. *Gaussian processes for machine learning*. Adaptive computation and machine learning. MIT Press, 2006.

[27] T. Ridnik, D. Kredo, and I. Friedman. Code generation with alphacodium: From prompt engineering to flow engineering. *arXiv preprint arXiv:2401.08500*, 2024.

[28] B. Roziere, J. Gehring, F. Gloeckle, S. Sootla, I. Gat, X. E. Tan, Y. Adi, J. Liu, T. Remez, J. Rapin, et al. Code llama: Open foundation models for code. *arXiv preprint arXiv:2308.12950*, 2023.

[29] B. Shahriari, K. Swersky, Z. Wang, R. P. Adams, and N. De Freitas. Taking the human out of the loop: A review of bayesian optimization. *Proceedings of the IEEE*, 104(1):148–175, 2016.

[30] D. Sobania, J. Petke, M. Briesch, and F. Rothlauf. A comparison of large language models and genetic programming for program synthesis. *IEEE Transactions on Evolutionary Computation*, 2024.

[31] C. Spiess, D. Gros, K. S. Pai, M. Pradel, M. R. I. Rabin, S. Jha, P. Devanbu, and T. Ahmed. Quality and trust in llm-generated code. *arXiv preprint arXiv:2402.02047*, 2024.

[32] N. Tao, A. Ventresque, V. Nallur, and T. Saber. Enhancing program synthesis with large language models using many-objective grammar-guided genetic programming. *Algorithms*, 17(7):287, 2024.

[33] Y. Tian, W. Yan, Q. Yang, Q. Chen, W. Wang, Z. Luo, and L. Ma. Codehalu: Code hallucinations in llms driven by execution-based verification. *arXiv preprint arXiv:2405.00253*, 2024.

[34] H. Touvron, T. Lavril, G. Izacard, X. Martinet, M.-A. Lachaux, T. Lacroix, B. Rozière, N. Goyal, E. Hambro, F. Azhar, et al. Llama: Open and efficient foundation language models. *arXiv preprint arXiv:2302.13971*, 2023.

[35] H. Touvron, L. Martin, K. Stone, P. Albert, A. Almahairi, Y. Babaei, N. Bashlykov, S. Batra, P. Bhargava, S. Bhosale, et al. Llama 2: Open foundation and fine-tuned chat models. *arXiv preprint arXiv:2307.09288*, 2023.

[36] P. Wang, R. Shin, X. Liu, Y. Jin, P. Sharma, N. Keskar, G. Fung, M. Naik, and S. Yu. Codex: A large-scale neural network model for code generation. *arXiv preprint arXiv:2106.01482*, 2021.

[37] X. Wang, J. Wei, D. Schuurmans, Q. Le, E. Chi, S. Narang, A. Chowdhery, and D. Zhou. Self-consistency improves chain of thought reasoning in language models. *arXiv preprint arXiv:2203.11171*, 2022.

[38] J. Wei, X. Wang, D. Schuurmans, M. Bosma, F. Xia, E. Chi, Q. V. Le, D. Zhou, et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in neural information processing systems*, 35:24824–24837, 2022.

[39] C. Yang, X. Wang, Y. Lu, H. Liu, Q. V. Le, D. Zhou, and X. Chen. Large language models as optimizers. In *The Twelfth International Conference on Learning Representations*, 2024.