ARCHCODE: Incorporating Software Requirements in Code Generation with Large Language Models

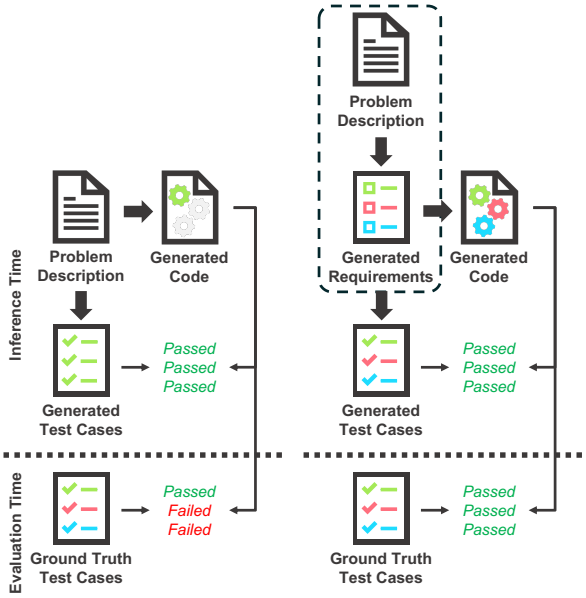
Hojae Han[♠◇] **Jaejin Kim**[♠◇] **Jaeseok Yoo**[♠] **Youngwon Lee**[♠◇] **Seung-won Hwang**[♠◇*]
[♠]Seoul National University, [◇]SNU-LG AI Research Center
{stovecat,jaejin.kim,jaeseok2.yoo,ludaya,seungwonh}@snu.ac.kr

Abstract

This paper aims to extend the code generation capability of large language models (LLMs) to automatically manage comprehensive software requirements from given textual descriptions. Such requirements include both functional (i.e. achieving expected behavior for inputs) and non-functional (e.g., time/space performance, robustness, maintainability) requirements. However, textual descriptions can either express requirements verbosely or may even omit some of them. We introduce ARCHCODE, a novel framework that leverages in-context learning to organize requirements observed in descriptions and to extrapolate unexpressed requirements from them. ARCHCODE generates requirements from given descriptions, conditioning them to produce code snippets and test cases. Each test case is tailored to one of the requirements, allowing for the ranking of code snippets based on the compliance of their execution results with the requirements. Public benchmarks show that ARCHCODE enhances to satisfy functional requirements, significantly improving Pass@k scores. Furthermore, we introduce HumanEval-NFR, the first evaluation of LLMs' non-functional requirements in code generation, demonstrating ARCHCODE's superiority over baseline methods. The implementation of ARCHCODE and the HumanEval-NFR benchmark are both publicly accessible.[1]

1 Introduction

Recent advancements in large language models (LLMs) have significantly improved code generation capabilities (Chen et al., 2021; Li et al., 2022; OpenAI, 2023). Although the primary goal for LLMs in this domain is to generate functionally correct code based on textual descriptions (Hendrycks et al., 2021; Austin et al., 2021; Chen et al., 2021; Li et al., 2022), real-world software development encompasses more than just functionality.



(a) Existing Approaches. (b) ARCHCODE (this work).

Figure 1: The ARCHCODE framework infers software requirements of correct code solution for a given textual description, then conditions them to generate code, as well as test cases for verification.

In software engineering, software requirements provide a detailed framework describing what a software system is intended to achieve (Chung et al., 2012), divided into two categories (Glinz, 2007):

- **Functional Requirements** (FRs) dictate the behavior and functionality, e.g., input/output conditions, desired behavior of code, etc.
- **Non-Functional Requirements** (NFRs) are attributes or constraints beyond functionality, e.g., time and space performance, robustness, maintainability, reliability, etc.

Despite the critical role of software requirements, considering these criteria has not been studied actively in previous code generation works, merely

* Corresponding author.
[1] https://github.com/ldilab/ArchCode

Proceedings of the 62nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers), pages 13520–13552

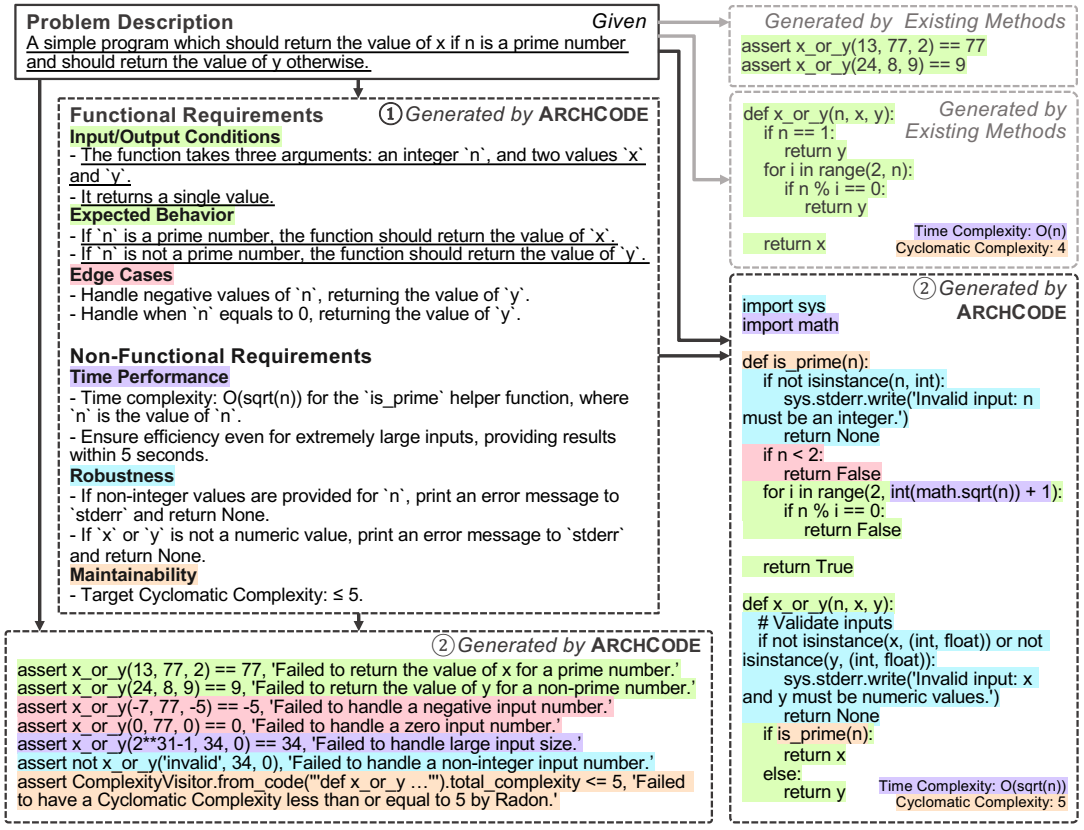


Figure 2: An illustrative example of code and test case generation. Existing approaches derive code and test cases directly from problem descriptions, often missing key requirements. ARCHCODE, in contrast, ① reformulates (underlined) and extrapolates (not underlined) requirements from these descriptions, then ② generates code and test cases to meet them comprehensively. Best viewed in color.

generating code directly from textual descriptions. However, textual descriptions might express requirements verbosely or even omit them. As illustrated in Figure 1a and 2 (upper right), this may result in code that neglects many desirable requirements. Code filtering based on generated test cases (Li et al., 2022; Chen et al., 2023; Huang et al., 2023) shares the same problem, as test cases often fail to cover a broader range of requirements. Consequently, the generated code might exhibit unexpected behaviors for valid inputs, ignoring FRs. Similarly, overlooking NFRs can result in time/space inefficiencies, potential system failures, or challenges in maintenance. Nevertheless, achieving conciseness in the textual descriptions of software requirements necessitates significant human effort (Perry and Wolf, 1992; Bass et al., 2003).

We introduce ARCHCODE, a novel framework that automatically incorporates software requirements from textual descriptions, then directs LLMs to align code and test case generation with those requirements, as illustrated in Figure 1b. Specifically, ARCHCODE leverages In-Context Learning (ICL; Kojima et al., 2022; Shao et al., 2023; Zhang et al., 2023c) for adaptability, utilizing LLMs' extensive reasoning abilities to learn within context, thereby avoiding costly parameter updates. For code generation, each in-context example comprises a triplet—a textual description, a list of software requirements (including both those expressed and unexpressed in the description), and corresponding code that satisfies all these requirements. For test case generation, we simply switch from code to test cases, each of which verifies a specific requirement. ARCHCODE prepends in-context examples to test descriptions, guiding LLMs to: 1) reformulate explicit requirements in descriptions, 2) deduce implicit requirements from their parametric knowledge, 3) generate code that fulfills these requirements, and 4) produce test cases for verifying each requirement, as shown in Figure 2.

We integrate ARCHCODE with *WizardCoder* (Luo et al., 2023) and GPT-3.5-Turbo (OpenAI, 2022), and assess the performance on HumanEval (Chen et al., 2021) and CodeContests (Li et al., 2022). The results confirm

that ARCHCODE notably outperforms existing techniques in terms of the satisfaction of FRs—surpassing GPT-4's Pass@1 score on both benchmarks and achieving new state-of-the-art on CodeContests. Moreover, we introduce HumanEval-NFR based on HumanEval, the first benchmark to evaluate NFRs alongside FRs, to confirm that ARCHCODE is also effective in pursuing NFRs.

Our main contributions are as follows:

- We propose ARCHCODE, a novel framework that leverages ICL to incorporate software requirements in code generation.
- ARCHCODE with GPT-3.5-Turbo surpasses GPT-4's Pass@1 scores on both HumanEval and CodeContests by 4.81%p and 10.45%p, while requiring $50\times$ smaller number of test cases to be generated compared to existing methods.
- We introduce HumanEval-NFR, the first code generation benchmark for NFR evaluation to confirm the effectiveness of ARCHCODE for NFR satisfaction.

2 Related Work

Despite the fact that LLMs recently have shown impressive capabilities in code generation, the majority of evaluations have focused solely on functional requirements (FRs; Hendrycks et al., 2021; Austin et al., 2021; Chen et al., 2021; Li et al., 2022).

Solely Targeting Functional Requirements Early research such as Feng et al. (2020), Chen et al. (2021), Brown et al. (2020), and Li et al. (2022) directly generates code from natural language descriptions, which may not fully capture all software requirements due to their vagueness or imperfections. Later studies (Jiang et al., 2023; Li et al., 2023; Zhang et al., 2023a) have targeted to better capture functional requirements by generating code-like outlines via in-context learning (ICL; Kojima et al., 2022; Shao et al., 2023; Zhang et al., 2023c). More recent methods enhance FR satisfaction through self-verification of the generated code: On one hand, *code filtering* utilizes 'over-generate-then-filter' strategies, where the filtering can be achieved either by predicting functional correctness without code execution (Inala et al., 2022; Ni et al., 2023; Zhang et al., 2023b), or execution with given (Shi et al., 2022b) or generated test cases (Li et al., 2022; Chen et al., 2023; Huang et al., 2023). On the other hand, *code refinement* iteratively reflects and refines the generated code to improve its functionality via execution results of current version of code with generated test cases (Shinn et al., 2023).

	FRs	NFRs
Self-planning (Jiang et al., 2023)	✓	✗
BRAINSTORM (Li et al., 2023)	✓	✗
ALGO (Zhang et al., 2023a)	✓	✗
CODERANKER (Inala et al., 2022)	✓	✗
LEVER (Ni et al., 2023)	✓	✗
Coder-Reviewer (Zhang et al., 2023b)	✓	✗
AlphaCode (Li et al., 2022)	✓	✗
MBR-EXEC (Shi et al., 2022a)	✓	✗
CODET (Chen et al., 2023)	✓	✗
REFLEXION (Shinn et al., 2023)	✓	✗
MPSC (Huang et al., 2023)	✓	✗
WizardCoder (Luo et al., 2023)	✓	△
PIE (Madaan et al., 2023)	✗	△
TITANFUZZ (Deng et al., 2023)	✗	△
FUZZ4ALL (Xia et al., 2023)	✗	△
ARCHCODE (this work)	✓	✓

Table 1: ARCHCODE is a novel code and test case generation framework that pursues the satisfaction of both FRs and NFRs. In NFRs column, △ denotes that only one or two NFRs were addressed in those works, whereas ARCHCODE addresses four different NFR categories, marked as ✓.

Targeting Narrow Scope of Non-Functional Requirements While much research covers FRs, few studies have addressed specific attributes of non-functional requirements (NFRs) such as reliability and robustness (Deng et al., 2023; Xia et al., 2023), or time/space performance (Madaan et al., 2023; Luo et al., 2023).

Our Distinction Table 1 summarizes the distinction of our framework compared with existing code generation approaches. To the best of our knowledge, ARCHCODE is the first study that employs ICL to systematically extract and interpret software requirements from descriptions, ensuring the generated code and test cases closely aligns with these requirements. In addition, we introduce HumanEval-NFR, a variant version of the HumanEval (Chen et al., 2021) benchmark that can assess the fulfillment of NFRs.

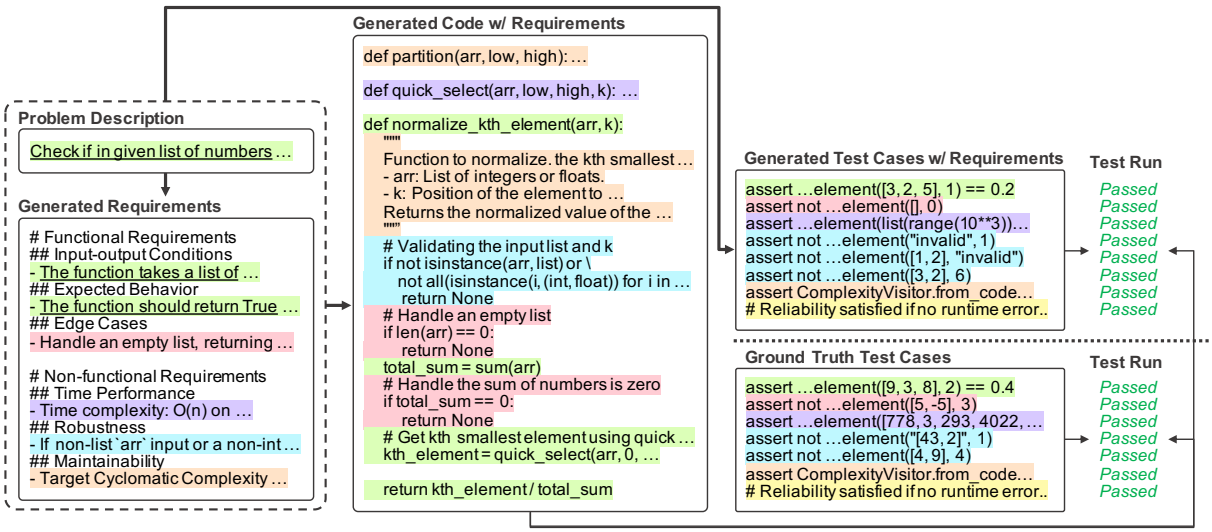


Figure 3: The overview of the ARCHCODE framework. Each color represents the subtype of software requirements. Underlined requirements are expressed in problem descriptions, whereas other requirements are inferred from descriptions by LLMs' parametric knowledge. Best viewed in color.

3 The ARCHCODE Framework

We propose ARCHCODE, a novel code generation framework that employs In-Context Learning (ICL) to LLMs, incorporating software requirements in code generation. As shown in Figure 3, ARCHCODE delineates software requirements from textual descriptions, generates code, then verifies it using custom test cases.[2]

Formally, given a problem space $\mathcal{P}$ and a code space $\mathcal{C}$, the code generation task is to build a function $\mathcal{F} : \mathcal{P} \rightarrow \mathcal{C}$ that maps each textual problem description $p \in \mathcal{P}$ into its corresponding code implementation $c \in \mathcal{C}$. ARCHCODE decomposes $\mathcal{F}$ by $\mathcal{F} = g \circ f$. $g : \mathcal{P} \rightarrow \mathcal{P} \times \mathcal{R}$ maps problems to problem-requirements pairs, and $f : \mathcal{P} \times \mathcal{R} \rightarrow \mathcal{C}$ generates code from the problem-requirements pairs, where $\mathcal{R}$ is a space of software requirements. The test case generation function $\mathcal{H} : \mathcal{P} \rightarrow \mathcal{T}$ is also decomposed by ARCHCODE into $\mathcal{H} = g \circ h$, where $\mathcal{T}$ is the space of test cases and $h : \mathcal{P} \times \mathcal{R} \rightarrow \mathcal{T}$.

3.1 Delineating Software Requirements

ARCHCODE leverages ICL to let an LLM generate a set of software requirements, either reformulated from a given textual description, or extrapolated from the description by the LLM's learnt parametric knowledge. Formally, given in-context examples of description-requirements pairs and the target description p, the LLM returns the list of software requirements

$$\hat{\mathbf{r}} = g([p'_1, \mathbf{r}'_1]; [p'_2, \mathbf{r}'_2]; ...; [p]), \quad (1)$$

where p'_i and $\mathbf{r}'_i = [r_i^1, r_i^2, ...]$ is the description and its corresponding requirement list of i-th example pair, each r_i^j in $\mathbf{r}'_i$ is a requirement, and $\hat{\mathbf{r}}$ is the list of generated requirements.

Specifically, based on the established classifications by Glinz (2007), we further break down FRs and NFRs into distinct categories that our study focuses on.

Target FRs Our approach narrows down FRs into three subtypes:

- **Input/Output Conditions**: Analogous to the preconditions and the postconditions in *Design by Contract* (Meyer, 1992), these define the desired functionality of the code by specifying valid inputs and expected outputs.
- **Expected Behavior**: Along with Input/Output Conditions, it explains the functionality of the target code by reformulating the description into a series of operations for valid inputs that are applicable in most general scenarios.
- **Edge Cases**: While this term generally comprises an array of corner cases, we restrict the scope to only consider valid inputs that necessitate distinct treatment. These include, for example, processing an empty list when the

[2]More details on input/output formats, in-context examples and hyperparameters are provided in Appendix A.

valid input type is a list, or considering '0' for non-negative integer inputs.

Target NFRs ARCHCODE considers NFRs that are both pivotal in real-world applications and feasible for assessment either through code execution or using existing metrics.

- **Time Performance**: Pertains to time-centric aspects like algorithmic time complexity or stipulated timeout conditions.
- **Robustness**: Ensures that code is resilient to invalid inputs (McConnell, 2004). For instance, a function designed for integer addition must prevent unforeseen or undesirable outcomes from the '+' operation, like mistakenly returning the concatenated string when given two strings.
- **Maintainability**: Considers factors that contribute to the ease of maintenance, such as reducing code complexity via code modularization (Magel et al., 1982), measured by cyclomatic complexity (McCabe, 1976).
- **Reliability**: Ensures that the code can handle errors gracefully, without causing system failures, thereby increasing the mean time between failures (McConnell, 2004).

3.2 Requirements-aware Generation

Upon obtaining software requirements $\hat{\mathbf{r}}$, ARCHCODE conditions $\hat{\mathbf{r}}$ with the given description p to generate code samples and test cases.[3] Specifically, ARCHCODE generates code $\hat{c}$ and test cases $\hat{\mathbf{t}}$ in a parallel manner:

$$\begin{aligned}\hat{c} &= f([p'_1, \mathbf{r}'_1, c'_1]; ...; [p, \hat{\mathbf{r}}]),\\ \hat{\mathbf{t}} &= h([p'_1, \mathbf{r}'_1, \mathbf{t}'_1]; ...; [p, \hat{\mathbf{r}}]),\end{aligned} \tag{2}$$

where c'_i and $\mathbf{t}'_i = [t_i^1, t_i^2, ...]$ are the code and the list of test cases of i-th example, and each t_i^j in $\mathbf{t}'_i$ is a test case corresponding to r_i^j in $\hat{\mathbf{r}}$.[4] We choose this parallel generation due to the potential pitfalls when these processes condition each other. We further discuss such pitfalls in Section 5.2.

[3]For the reliability category, we uniquely assess code reliability by checking for runtime errors with various test cases, instead of generating specific ones.

[4]For an intuitive explanation, we describe how a single test case is tailored to a requirement. However, in real implementation, ARCHCODE utilizes multiple generated test cases to confirm each requirement, as explained in Appendix E.

3.3 Pursuing Requirements Satisfaction

To ensure the conformance of the generated code snippet $\hat{c}$ with the specified requirements $\hat{\mathbf{r}}$, ARCHCODE executes $\hat{c}$ against the generated test cases $\hat{\mathbf{t}}$ tailored to one of the requirements in $\hat{\mathbf{r}}$:

$$s = \texttt{EXEC}\left(\hat{c}, \hat{t}\right), \tag{3}$$

where $s \in \{0, 1\}$ is a binary result from a code execution function EXEC, and $\hat{t}$ is one of the generated test cases in $\hat{\mathbf{t}}$, matching $\hat{r}$ in $\hat{\mathbf{r}}$. To return the satisfactory code towards $\hat{\mathbf{r}}$, ARCHCODE conducts code filtering. To rank each code in relation to $\hat{\mathbf{r}}$, our framework calculates a weighted sum of the scores s from each $\hat{t}$, with the option to assign higher weights to preferred requirements. Adjusting those weights to tailor the scoring process is discussed in more detail in Section 5.3.

4 Experiments

We evaluate ARCHCODE's effectiveness using three benchmarks, categorized into two types: 1) A novel benchmark for assessing both FR and NFR satisfaction; 2) Two public benchmarks aimed at FR evaluation, facilitating comparison of ARCHCODE with existing baselines. For the former, we introduce HumanEval-NFR for comprehensive NFR assessment, overcoming the conventional focus on FR alone. For the latter, we explore two code modalities: 1) function-level and 2) competition-level code generation.

4.1 Experimental Setup

We evaluate the effectiveness of ARCHCODE on code generation with LLMs. Throughout the experiments, we used GPT-3.5-Turbo-16k (OpenAI, 2022) as the backbone LLMs for generating code, software requirements, test cases, etc. More details can be found in Appendix A.

Evaluation Metrics We mainly consider the widely used Pass@$k := \mathbb{E}_{\text{Problems}}[1 - \frac{\binom{n-c}{k}}{\binom{n}{k}}]$ (Chen et al., 2021) metric for evaluation, which is the unbiased estimator of the probability that the code generation system would have passed a problem if it were given k chances to sample c correct code snippets among n samples. Adhering to Chen et al. (2023), when applying code filtering, we denote the existence of passed code among the k filtered samples.[5]

[5]While this metric is sometimes referred to as $n@k$—the pass ratio of filtered n samples from k—we avoid this notation

	All		Time Perf.		Robustness		Maintainability		Reliability	
Pass@k	k=1	5	k=1	5	k=1	5	k=1	5	k=1	5
GPT-3.5-Turbo	2.62	10.03	53.48	65.75	4.21	14.55	53.23	68.38	20.98	36.72
+ CoT	5.00	12.08	50.00	66.03	7.32	17.67	44.33	62.00	45.49	66.83
CodeT	3.03	10.03	58.50	67.31	4.61	14.52	**57.80**	**68.50**	47.62	**74.90**
ArchCode	**25.19**	**27.33**	**62.86**	**69.70**	**40.86**	**42.72**	56.43	62.23	**68.53**	74.67

Table 2: Experimental results on HumanEval-NFR. Each column states the evaluation category of NFRs. Boldface and underline denote the 1st and 2nd highest scores, respectively. MPSC is omitted as it is publicly unavailable.

Method	CoT	Code Filtering	NFRs	Pass@k HumanEval			Pass@k CodeContests		
				k=1	2	5	k=1	2	5
CodeRanker†	✗	✓	✗	32.3	-	61.6	-	-	-
WizardCoder 34B†	✗	✗	△	73.2	-	-	-	-	-
GPT-4‡	✗	✗	✗	81.55	86.39	**90.49**	6.07	8.23	11.67
GPT-3.5-Turbo	✗	✗	✗	73.17	80.79	86.99	4.79	7.02	10.06
+ CoT	✓	✗	✗	72.99	79.58	83.95	5.82	8.57	13.53
BrainStorm†	✓	✗	✗	-	-	-	7.0	-	14.7
Algo†	✓	✗	✗	-	-	-	12.00	12.00	-
MBR-Exec‡	✗	✓	✗	72.96	76.47	79.00	8.25	8.87	9.08
CodeT‡	✗	✓	✗	78.05	78.05	78.30	9.92	10.18	10.30
MPSC†	✓	✓	✗	85.37	86.60	86.35	14.39	**17.16**	**17.76**
ArchCode (Ours)	✓	✓	✓	**86.36**	**88.62**	90.48	**16.52**	16.67	17.37

Table 3: Experimental results on HumanEval and CodeContests. Daggers† denote the values are directly sourced from the respective original works, and the results with double daggers‡ are from Huang et al. (2023). The results for BrainStorm, Algo, MBR-Exec, CodeT, and MPSC are based on GPT-3.5-Turbo. The empty results for CodeRanker, BrainStorm, and Algo are due to reproducibility issues, as both the checkpoint and the full training data for each method are publicly unavailable.

Baselines Throughout the benchmarks, we consider three baselines: GPT-3.5-Turbo and its CoT prompting applied version of Self-planning (Jiang et al., 2023), and CodeT (Chen et al., 2023). For both HumanEval and CodeContests, we further use three code filtering methods—CodeRanker (Inala et al., 2022), MBR-Exec (Shi et al., 2022b), and MPSC (Huang et al., 2023)—along with *WizardCoder* 34B (Luo et al., 2023) and GPT-4 (OpenAI, 2023). For CodeContests, we additionally compare with two CoT methods: BrainStorm (Li et al., 2023) and Algo (Zhang et al., 2023a).

4.2 HumanEval-NFR: Embracing NFR Evaluation

We introduce HumanEval-NFR benchmark, which is specifically designed to assess NFR satisfaction. It is an extension of HumanEval that additionally covers four NFR categories, chosen for their suitability for evaluation, through code execution using annotated test cases or automated assessment using existing metrics. Details on the annotation process and metrics we used are provided in Appendix B.

Table 2 presents that ArchCode outperforms all baseline methods across various NFR categories except for maintainability. Our conjecture is that, as which NFR categories to prioritize is uninformed in this experiment, ArchCode's consideration of all NFRs could potentially impede maintainability due to the influence of other categories. We study the informed case of optimizing specific categories in Section 5.3. Across all approaches, satisfying the robustness category appears to be more difficult compared to other NFR categories, for which we provide further discussion in Appendix G.

Notably, ArchCode is desirable for evaluating all NFRs at once (i.e. All), outperforming CodeT with 22.16% of Pass@1.

as the interpretation of k differs from that in Pass@k.

4.3 HumanEval and CodeContests: Public Benchmarks for FR Evaluation

We additionally report results on two popular code generation benchmarks targetting functional correctness. HumanEval (Chen et al., 2021) is a hand-crafted test benchmark with 164 programming problems along with public and hidden test cases. CodeContests (Li et al., 2022) consists of 13k/113/165 instances of train/valid/test data collected from multiple code competition websites. While HumanEval tests the model's capability to implement rather simpler functions without errors, the competitive programming oriented nature of CodeContests often requires more complex form of reasoning such as algorithmic reasoning. Each of these addresses different aspect of industrial software: the former is related to solving each of the simpler tasks composing larger and complex projects while the latter focuses on the logical and algorithmic perspective of software development.

In Table 3, ARCHCODE consistently outperforms the baseline methods. Specifically, on both benchmarks, ARCHCODE leveraging GPT-3.5-Turbo, exceeds GPT-4's performance by a substantial margin of 4.81%p and 10.45%p in terms of Pass@1. In comparison with *WizardCoder* 34B—a baseline that partially incorporates NFR considerations during the finetuning phase—ARCHCODE, which covers NFRs more comprehensively, achieves significantly higher performance. In CodeContests, while our custom GPT-3.5-Turbo + CoT prompting baseline is outdone by the state-of-the-art CoT methods BRAINSTORM and ALGO, the application of ARCHCODE outperforms both approaches, setting new state-of-the-art of Pass@1. We also compare ARCHCODE with MPSC, a very recent baseline. Notably, ARCHCODE surpasses MPSC in all Pass@k metrics on HumanEval and Pass@1 on CodeContests, while ARCHCODE is much more cost-efficient. We provide further discussion on computational costs in Section 5.1.

5 Analysis and Discussion

5.1 Efficiency and Effectiveness of Requirement-aware Test Case Generation

Efficiency In code filtering, a crucial step involves minimizing the number of generated test cases to reduce computational and time costs for code execution. As shown in Figure 4, existing approaches such as MPSC and CODET requires to generate hundreds of test cases for performance.

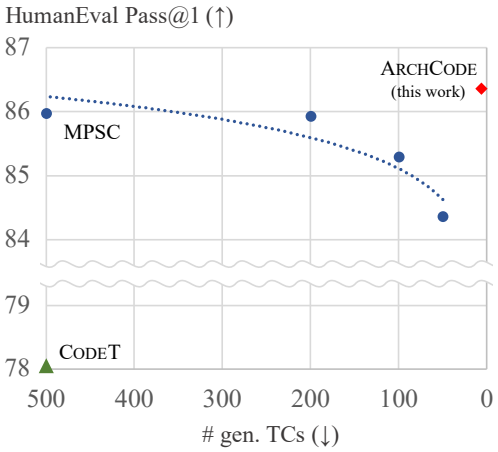


Figure 4: Pass@1 versus average number of test cases needed per problem on HumanEval. ARCHCODE (◆) achieves the highest Pass@1 score with significantly less number of generated test cases. All values are obtained from GPT-3.5-Turbo. The values for MPSC (●) and CODET (▲) are from Huang et al. (2023). Best viewed in color.

Test Case Generation Method	Code Gen. Method ARCHCODE		
	k=1	2	5
None	6.73	9.79	14.63
CODET	11.09	13.59	17.18
CODET (w/o clustering)	13.16	14.14	16.48
ARCHCODE	**16.52**	**16.67**	**17.37**

Table 4: Code filtering results with different test case generation methods on CodeContests, while the code generation method is fixed to ARCHCODE. GPT-3.5-Turbo is used as the backbone model. MPSC is omitted as it is publicly unavailable.

In contrast, ARCHCODE targeting diverse requirement categories shows the best performance while **significantly improving the efficiency by generating 50x smaller number of test cases.**

Effectiveness Tables 4, 5, and 6 compare two test case generation methods, the naive way (CODET) and ARCHCODE. With the same code generation and filtering strategy applied, the latter generally outperforms the former with large margins, demonstrating the effectiveness of leveraging generated requirements to optimize test case generation. Meanwhile, ARCHCODE yielded comparable results to CODET without the use of clustering on HumanEval. We conjecture that for simpler benchmarks like HumanEval, CODET's approach of generating 'general' test cases suffices. While

Test Case Generation Method	Code Generation Method: GPT-3.5-Turbo			GPT-3.5-Turbo + CoT			ARCHCODE		
	k=1	2	5	k=1	2	5	k=1	2	5
None	2.62	4.91	10.03	5.00	7.90	12.08	15.85	20.23	24.83
CODET	3.03	4.83	10.03	5.50	8.05	12.04	17.00	19.49	24.69
CODET (w/o clustering)	2.86	5.02	10.30	4.69	7.21	11.72	16.09	20.24	25.07
ARCHCODE	**13.87**	**14.53**	**14.63**	**13.82**	**14.46**	**14.52**	**25.81**	**26.84**	**27.20**

Table 5: Code filtering results with different test case generation methods on HumanEval-NFR (All). GPT-3.5-Turbo is used as the backbone model. MPSC is omitted as it is publicly unavailable.

Test Case Generation Method	Code Gen. Method: ARCHCODE		
	k=1	2	5
None	75.06	81.83	87.95
CODET	79.92	87.63	**91.00**
CODET (w/o clustering)	**86.40**	88.21	90.66
ARCHCODE	86.36	**88.62**	90.48

Table 6: Code filtering results with different test case generation methods on HumanEval, while the code generation method is fixed to ARCHCODE. GPT-3.5-Turbo is used as the backbone model. MPSC is omitted as it is publicly unavailable.

CODET focuses on general test cases which are likely to have limited coverage, **ARCHCODE distinctly promotes a diverse set of test cases targeting various requirement (sub)types.**

5.2 Conditioning Code Generation on Test Cases

In contrast to our approach of generating code and test cases in parallel and then applying subsequent postprocess mechanisms such as filtering, one can also consider conditioning the code generation on test cases,[6] taking inspiration from the Test-Driven Development (TDD; Beck, 2022) methodology. Table 7 shows results consistent with those reported in Chen et al. (2023), indicating marginal improvement in performance is observed when conditioning code generation on the ground-truth and generated test cases, while **incorporating software requirements through ARCHCODE effectively boosts the score,** even without code filtering. This suggests the overhead from introducing new sequential dependency in the generation process might not be worth the additional costs incurred.

[6]One may also consider the opposite direction of generating the test cases conditioned on the generated code, which we do not visit in this paper.

Method	Pass@k		
	k=1	2	5
GPT-3.5-Turbo	73.17	80.79	86.99
+ Gold Test Cases	73.60	80.93	87.24
+ Generated Test Cases	73.66	80.54	86.53
ARCHCODE − Filtering	**75.06**	**81.83**	**87.95**

Table 7: Results on HumanEval with generating code conditioned additionally on test cases. While incurring sequential dependency and increased latency, TDD-like conditioning brings marginal improvement, as opposed to our method being effective even without filtering.

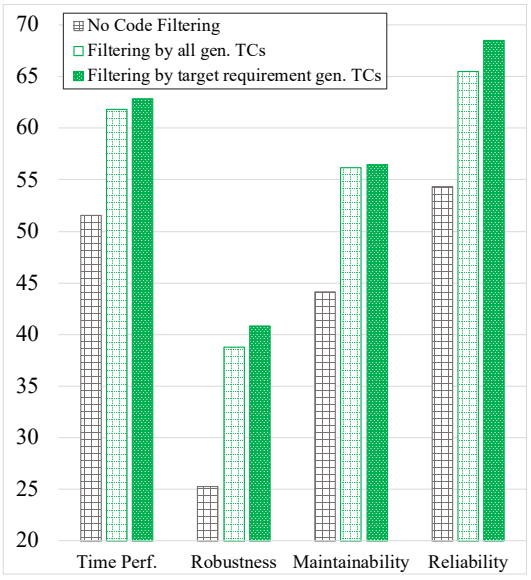


Figure 5: Pass@1 score of ARCHCODE for each requirement category in HumanEval-NFR. Using dedicated test cases for filtering consistently outperforms blindly using all test cases. Best viewed in color.

5.3 Preference over Requirements

As mentioned before, ARCHCODE can be informed of any user preferences over the software requirements at code filtering time—after several code candidates have been generated and awaiting to be ranked. Figure 5 presents the Pass@1 scores

for each NFR category in the HumanEval-NFR benchmark, with different code filtering strategies applied. Using targeted test cases for reranking yields higher pass rates for that specific software requirement than using all test cases does.

Another approach to incorporate user preference over the requirements is to consider a subset of requirements when generating code or test cases, or to put emphasis on a subset of requirements by editing the prompt while presenting all requirements to the model. We present detailed analyses for such scenarios in Appendix F.

5.4 ARCHCODE under Diverse Settings

Here we provide empirical results suggesting that ARCHCODE generalizes well to other models, datasets, etc. than those considered in the main experiments.

Open-source LLMs First, we showcase ARCHCODE combined with a relatively smaller model, namely *WizardCoder* 7B. Table 8 indicates that applying ARCHCODE with the said backbone model leads to a notable 15.67%p improvement in Pass@1 on HumanEval, while incorporating in-context learning directly into *WizardCoder* 7B itself has negative impacts. Note that this observation is consistent with prior findings such as that in Yuan et al. (2023), that instruction tuning might compromise in-context learning capabilities of LLMs; *WizardCoder* 7B is an instruction-tuned model based on CODELLAMA 7B.

Meanwhile, in practical settings, diverse LLMs offer complementary benefits in terms of cost-performance trade-off, and thus mixing two models has been a conventional aproach to explore cost-performance design space (Sun et al., 2023; Wang et al., 2023). ARCHCODE$_{\text{MIX}}$ shown in Tables 8 and 9 similarly capitalizes on this space by directing most of the generation calls to affordable LLMs, while selectively delegating the part requiring the most of the reasoning capabilities to stronger ones.

Other Programming Languages We also extend the evaluation of ARCHCODE to the task of Java code generation, using the MultiPL-E (Cassano et al., 2022) benchmark and the backbone model SantaCoder 1B (Allal et al., 2023). To address the rather limited capacity of a smaller model, we further applied sparse fine-tuning (Ansell et al. (2022); SFT) on a public Java train set. We provide more details in Appendix A.1. The results in Table 9 demonstrate the effectiveness of the proposed method in generating Java code, supporting that our method is generally applicable to programming languages other than Python.

Method	Pass@k		
	k=1	2	5
w/o ICL			
WizardCoder 7B	48.54	60.08	72.48
w/ ICL			
WizardCoder 7B	43.35	53.74	65.42
+ CoT	42.01	53.74	67.27
ARCHCODE	64.21	67.72	72.84
ARCHCODE$_{\text{MIX}}$	**68.33**	**71.36**	**74.44**

Table 8: Experimental results using *WizardCoder* 7B for code generation on HumanEval. 'w/ ICL' means that 1-shot in-context learning is employed. 'ARCHCODE$_{\text{MIX}}$' indicates that code filtering is applied with test cases generated by ARCHCODE using GPT-3.5-Turbo.

Method	Pass@1
SantaCoder 1B	15.00
+ SFT	18.16
ARCHCODE$_{\text{MIX}}$	**24.61**

Table 9: Experimental results on MultiPL-E java. 'SFT' denotes sparse fine-tuning. 'ARCHCODE$_{\text{MIX}}$' indicates that code filtering is applied with test cases generated by ARCHCODE using GPT-3.5-Turbo.

6 Conclusion

We proposed ARCHCODE, a framework incorporating software requirements from textual descriptions for LLM-based code generation. This systematic approach not only identifies these requirements but also harnesses them to guide the code generation process. The verification of code snippets with the generated test cases tailored to each requirement provides a robust validation layer for the alignment with detected requirements. On HumanEval and CodeContests, ARCHCODE with GPT-3.5-Turbo exceeded GPT-4's performance by 4.81%p and 10.45%p of Pass@1. ARCHCODE requires 50x less generated test cases compared to MPSC and CODET, while outperforming them. In addition, we introduced a new benchmark named HumanEval-NFR for evaluating how well LLMs can pursue non-functional requirements in code generation task. Further analysis shows the pertinence of parallel generation of code and test case, and the efficiency and the effectiveness of ARCHCODE's requirement-aware test case generation.

Acknowledgment

This work was supported by LG AI Research, and partly supported by Electronics and Telecommunications Research Institute (ETRI) grant funded by ICT R&D program of MSIT/IITP (2022-0-00995, Automated reliable source code generation from natural language descriptions).

Limitations

ARCHCODE leverages in-context learning as a tool to integrate both functional and non-functional requirements in the processes of code and test case generation. We did not studied prompt engineering and devising more sophisticated in-context examples which is beyond the scope of this work.

ARCHCODE encompassed three functional and four non-functional requirements, aligning with the established taxonomy within software engineering literature (Glinz, 2007). However, the potential for future work lies in addressing more complex and varied requirements involving larger pieces of code, as well as accommodating changes in software requirements over time.

Lastly, as ARCHCODE relies on generated requirements to guide subsequent code and test case generation process, although qualitative analysis suggests its impact could be limited in practice, additional measures to mitigate cascading errors via human intervention or self-correction by LLMs, etc. (Shinn et al., 2023; Wang et al., 2023; Yao et al., 2023; Chen et al., 2024) can be necessitated.

Ethical and Social Implications

ARCHCODE leverages LLMs to automatically generate software requirements, code, and test cases, thereby enhancing productivity and reducing manual labor for developers. However, to maximize these advantages while addressing potential risks, such as the creation of code with safety or security vulnerabilities as discussed in Chen et al. (2021), careful consideration is essential. Strategies to mitigate these risks include establishing default requirements for desired outcomes, delineating the permissible scope of generated code, and ensuring that the code remains within its authorized boundaries.

References

Loubna Ben Allal, Raymond Li, Denis Kocetkov, Chenghao Mou, Christopher Akiki, Carlos Munoz Ferrandis, Niklas Muennighoff, Mayank Mishra, Alex Gu, Manan Dey, et al. 2023. Santacoder: don't reach for the stars! *arXiv preprint arXiv:2301.03988*.

Alan Ansell, Edoardo Ponti, Anna Korhonen, and Ivan Vulić. 2022. Composable sparse fine-tuning for cross-lingual transfer. In *Proceedings of the 60th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1778–1796, Dublin, Ireland. Association for Computational Linguistics.

Jacob Austin, Augustus Odena, Maxwell I. Nye, Maarten Bosma, Henryk Michalewski, David Dohan, Ellen Jiang, Carrie J. Cai, Michael Terry, Quoc V. Le, and Charles Sutton. 2021. Program synthesis with large language models. *CoRR*, abs/2108.07732.

Len Bass, Paul Clements, and Rick Kazman. 2003. *Software architecture in practice*. Addison-Wesley Professional.

Kent Beck. 2022. *Test driven development: By example*. Addison-Wesley Professional.

Tom B Brown, Benjamin Mann, Nick Ryder, Melanie Subbiah, Jared Kaplan, Prafulla Dhariwal, Arvind Neelakantan, Pranav Shyam, Girish Sastry, Amanda Askell, et al. 2020. Language models are few-shot learners. *arXiv preprint arXiv:2005.14165*.

Federico Cassano, John Gouwar, Daniel Nguyen, Sydney Nguyen, Luna Phipps-Costin, Donald Pinckney, Ming-Ho Yee, Yangtian Zi, Carolyn Jane Anderson, Molly Q Feldman, et al. 2022. Multipl-e: A scalable and extensible approach to benchmarking neural code generation. *arXiv preprint arXiv:2208.08227*.

Bei Chen, Fengji Zhang, Anh Nguyen, Daoguang Zan, Zeqi Lin, Jian-Guang Lou, and Weizhu Chen. 2023. Codet: Code generation with generated tests. In *The Eleventh International Conference on Learning Representations, ICLR 2023, Kigali, Rwanda, May 1-5, 2023*. OpenReview.net.

Mark Chen, Jerry Tworek, Heewoo Jun, Qiming Yuan, Henrique Pondé de Oliveira Pinto, Jared Kaplan, Harrison Edwards, Yuri Burda, Nicholas Joseph, Greg Brockman, Alex Ray, Raul Puri, Gretchen Krueger, Michael Petrov, Heidy Khlaaf, Girish Sastry, Pamela Mishkin, Brooke Chan, Scott Gray, Nick Ryder, Mikhail Pavlov, Alethea Power, Lukasz Kaiser, Mohammad Bavarian, Clemens Winter, Philippe Tillet, Felipe Petroski Such, Dave Cummings, Matthias Plappert, Fotios Chantzis, Elizabeth Barnes, Ariel Herbert-Voss, William Hebgen Guss, Alex Nichol, Alex Paino, Nikolas Tezak, Jie Tang, Igor Babuschkin, Suchir Balaji, Shantanu Jain, William Saunders, Christopher Hesse, Andrew N. Carr, Jan Leike, Joshua Achiam, Vedant Misra, Evan Morikawa, Alec Radford, Matthew Knight, Miles Brundage, Mira Murati, Katie Mayer, Peter Welinder, Bob McGrew, Dario Amodei, Sam McCandlish, Ilya Sutskever, and Wojciech Zaremba. 2021. Evaluating large language models trained on code. *CoRR*, abs/2107.03374.

Xinyun Chen, Maxwell Lin, Nathanael Schärli, and Denny Zhou. 2024. Teaching large language models to self-debug. In *The Twelfth International Conference on Learning Representations*.

Lawrence Chung, Brian A Nixon, Eric Yu, and John Mylopoulos. 2012. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media.

Yinlin Deng, Chunqiu Steven Xia, Haoran Peng, Chenyuan Yang, and Lingming Zhang. 2023. Large language models are zero-shot fuzzers: Fuzzing deep-learning libraries via large language models. In *Proceedings of the 32nd ACM SIGSOFT International Symposium on Software Testing and Analysis*, pages 423–435.

Zhangyin Feng, Daya Guo, Duyu Tang, Nan Duan, Xiaocheng Feng, Ming Gong, Linjun Shou, Bing Qin, Ting Liu, Daxin Jiang, and Ming Zhou. 2020. CodeBERT: A pre-trained model for programming and natural languages. In *Findings of the Association for Computational Linguistics: EMNLP 2020*, pages 1536–1547, Online. Association for Computational Linguistics.

Martin Glinz. 2007. On non-functional requirements. In *15th IEEE international requirements engineering conference (RE 2007)*, pages 21–26. IEEE.

Daniel Gross and Eric Yu. 2001. From non-functional requirements to design through patterns. *Requirements Engineering*, 6:18–36.

Dan Hendrycks, Steven Basart, Saurav Kadavath, Mantas Mazeika, Akul Arora, Ethan Guo, Collin Burns, Samir Puranik, Horace He, Dawn Song, and Jacob Steinhardt. 2021. Measuring coding challenge competence with APPS. In *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks 1, NeurIPS Datasets and Benchmarks 2021, December 2021, virtual*.

Ari Holtzman, Jan Buys, Li Du, Maxwell Forbes, and Yejin Choi. 2020. The curious case of neural text degeneration. In *International Conference on Learning Representations*.

Baizhou Huang, Shuai Lu, Weizhu Chen, Xiaojun Wan, and Nan Duan. 2023. Enhancing large language models in coding through multi-perspective self-consistency. *arXiv preprint arXiv:2309.17272*.

Jeevana Priya Inala, Chenglong Wang, Mei Yang, Andres Codas, Mark Encarnación, Shuvendu K Lahiri, Madanlal Musuvathi, and Jianfeng Gao. 2022. Fault-aware neural code rankers. In *Advances in Neural Information Processing Systems*.

Xue Jiang, Yihong Dong, Lecheng Wang, Zheng Fang, Qiwei Shang, Ge Li, Zhi Jin, and Wenpin Jiao. 2023. Self-planning code generation with large language models.

Takeshi Kojima, Shixiang Shane Gu, Machel Reid, Yutaka Matsuo, and Yusuke Iwasawa. 2022. Large language models are zero-shot reasoners. *Advances in neural information processing systems*, 35:22199–22213.

Reto Krummenacher, Elena Paslaru Bontas Simperl, and Dieter Fensel. 2007. Towards scalable information spaces. *New Forms of Reasoning for the Semantic Web*, 291.

Xin-Ye Li, Jiang-Tian Xue, Zheng Xie, and Ming Li. 2023. Think outside the code: Brainstorming boosts large language models in code generation. *arXiv preprint arXiv:2305.10679*.

Yujia Li, David Choi, Junyoung Chung, Nate Kushman, Julian Schrittwieser, Rémi Leblond, Tom Eccles, James Keeling, Felix Gimeno, Agustin Dal Lago, Thomas Hubert, Peter Choy, Cyprien de Masson d'Autume, Igor Babuschkin, Xinyun Chen, Po-Sen Huang, Johannes Welbl, Sven Gowal, Alexey Cherepanov, James Molloy, Daniel J. Mankowitz, Esme Sutherland Robson, Pushmeet Kohli, Nando de Freitas, Koray Kavukcuoglu, and Oriol Vinyals. 2022. Competition-level code generation with alphacode. *Science*, 378(6624):1092–1097.

Ziyang Luo, Can Xu, Pu Zhao, Qingfeng Sun, Xiubo Geng, Wenxiang Hu, Chongyang Tao, Jing Ma, Qingwei Lin, and Daxin Jiang. 2023. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*.

Aman Madaan, Alexander Shypula, Uri Alon, Milad Hashemi, Parthasarathy Ranganathan, Yiming Yang, Graham Neubig, and Amir Yazdanbakhsh. 2023. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*.

Kenneth Magel, Raymond Michael Kluczny, Warren A Harrison, and Arlan R Dekock. 1982. Applying software complexity metrics to program maintenance.

Thomas J McCabe. 1976. A complexity measure. *IEEE Transactions on software Engineering*, (4):308–320.

Steve McConnell. 2004. Code complete second edition.

Bertrand Meyer. 1992. Applying "design by contract". *Computer*, 25(10):40–51.

Ana Moreira, Awais Rashid, and João Araújo. 2005. Multi-dimensional separation of concerns in requirements engineering. In *13th IEEE International Conference on Requirements Engineering (RE'05)*, pages 285–296. IEEE.

Ansong Ni, Srini Iyer, Dragomir Radev, Veselin Stoyanov, Wen-tau Yih, Sida Wang, and Xi Victoria Lin. 2023. Lever: Learning to verify language-to-code generation with execution. In *International Conference on Machine Learning*, pages 26106–26128. PMLR.

Erik Nijkamp, Bo Pang, Hiroaki Hayashi, Lifu Tu, Huan Wang, Yingbo Zhou, Silvio Savarese, and Caiming Xiong. 2023. Codegen: An open large language model for code with multi-turn program synthesis. In *International Conference on Learning Representations*.

OpenAI. 2022. Chatgpt: Optimizing language models for dialogue. openai.

OpenAI. 2023. Gpt-4 technical report.

Dewayne E Perry and Alexander L Wolf. 1992. Foundations for the study of software architecture. *ACM SIGSOFT Software engineering notes*, 17(4):40–52.

Henry Gordon Rice. 1953. Classes of recursively enumerable sets and their decision problems. *Transactions of the American Mathematical society*, 74(2):358–366.

Zhihong Shao, Yeyun Gong, Yelong Shen, Minlie Huang, Nan Duan, and Weizhu Chen. 2023. Synthetic prompting: Generating chain-of-thought demonstrations for large language models. In *Proceedings of the 40th International Conference on Machine Learning*, ICML'23. JMLR.org.

Freda Shi, Daniel Fried, Marjan Ghazvininejad, Luke Zettlemoyer, and Sida I. Wang. 2022a. Natural language to code translation with execution. In *Proceedings of the 2022 Conference on Empirical Methods in Natural Language Processing*, pages 3533–3546, Abu Dhabi, United Arab Emirates. Association for Computational Linguistics.

Zhengxiang Shi, Yue Feng, and Aldo Lipani. 2022b. Learning to execute actions or ask clarification questions. In *Findings of the Association for Computational Linguistics: NAACL 2022*, pages 2060–2070, Seattle, United States. Association for Computational Linguistics.

Noah Shinn, Federico Cassano, Ashwin Gopinath, Karthik Narasimhan, and Shunyu Yao. 2023. Reflexion: Language agents with verbal reinforcement learning. In *Advances in Neural Information Processing Systems*.

Weiwei Sun, Lingyong Yan, Xinyu Ma, Shuaiqiang Wang, Pengjie Ren, Zhumin Chen, Dawei Yin, and Zhaochun Ren. 2023. Is ChatGPT good at search? investigating large language models as re-ranking agents. In *Proceedings of the 2023 Conference on Empirical Methods in Natural Language Processing*, pages 14918–14937, Singapore. Association for Computational Linguistics.

Guanzhi Wang, Yuqi Xie, Yunfan Jiang, Ajay Mandlekar, Chaowei Xiao, Yuke Zhu, Linxi Fan, and Anima Anandkumar. 2023. Voyager: An open-ended embodied agent with large language models. *arXiv preprint arXiv:2305.16291*.

Arthur Henry Watson, Dolores R Wallace, and Thomas J McCabe. 1996. *Structured testing: A testing methodology using the cyclomatic complexity metric*, volume 500. US Department of Commerce, Technology Administration, National Institute of ....

Chunqiu Steven Xia, Matteo Paltenghi, Jia Le Tian, Michael Pradel, and Lingming Zhang. 2023. Universal fuzzing via large language models. *arXiv preprint arXiv:2308.04748*.

Shunyu Yao, Dian Yu, Jeffrey Zhao, Izhak Shafran, Thomas L. Griffiths, Yuan Cao, and Karthik R Narasimhan. 2023. Tree of thoughts: Deliberate problem solving with large language models. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Zhiqiang Yuan, Junwei Liu, Qiancheng Zi, Mingwei Liu, Xin Peng, and Yiling Lou. 2023. Evaluating instruction-tuned large language models on code comprehension and generation. *arXiv preprint arXiv:2308.01240*.

Kexun Zhang, Danqing Wang, Jingtao Xia, William Yang Wang, and Lei Li. 2023a. ALGO: Synthesizing algorithmic programs with generated oracle verifiers. In *Thirty-seventh Conference on Neural Information Processing Systems*.

Tianyi Zhang, Tao Yu, Tatsunori Hashimoto, Mike Lewis, Wen-tau Yih, Daniel Fried, and Sida Wang. 2023b. Coder reviewer reranking for code generation. In *International Conference on Machine Learning*, pages 41832–41846. PMLR.

Zhuosheng Zhang, Aston Zhang, Mu Li, and Alex Smola. 2023c. Automatic chain of thought prompting in large language models. In *The Eleventh International Conference on Learning Representations*.

A Implementation Details

We used `gpt-3.5-turbo-16k` (OpenAI, 2022) as the backbone LLM for most of the experiments, with ICL and nucleus sampling (Holtzman et al., 2020) with $p = 0.95$ and temperature $T = 0.8$ following Chen et al. (2021); Nijkamp et al. (2023); Chen et al. (2023). We used different in-context examples for each benchmark: a single HumanEval-style (problem description, code) pair from Li et al. (2023) for HumanEval-NFR, eight pairs from the training set of the MBPP (Austin et al., 2021) benchmark for HumanEval (Chen et al., 2021). For CodeContests (Li et al., 2022) we used a single pair from the train set.

To apply CoT prompting (Kojima et al., 2022; Shao et al., 2023; Zhang et al., 2023c), as the state-of-the-art methods BRAINSTORM (Li et al., 2023) and ALGO (Zhang et al., 2023a) are publicly unavailable, we generated the reasoning chains of code outline by using self-planning (Jiang et al., 2023). However, directly using the reasoning chains provided by Self-planning can result in data contamination on HumanEval because these chains are based on the test examples. Thus, rather using them directly, we utilized them to generate the reasoning chains for the aforementioned in-context examples, then used the generated reasoning chains for ICL.

ARCHCODE uses three reasoning chains when generating code: the initial program outline (the same reasoning chains as in GPT-3.5-Turbo + CoT), requirements described in Subsection 3.1 and Appendix D, and the final program outline—the revised version of the initial program outline, modified to meet the requirements.

We generated $n = 10$ code samples for every problem in the benchmarks. To enhance the diversity of the generated code, we employed nucleus sampling to produce n initial program outlines induced from Self-planning. The rest of the reasoning chains were concurrently generated using greedy sampling, culminating in a total of n final code outputs.

Our implementation is largely based on the LangChain library.[7] Regarding the execution and evaluation of the generated code, we modified some code from the CodeEval repository[8] which is available on Huggingface.

[7] https://github.com/langchain-ai/langchain

[8] https://huggingface.co/spaces/evaluate-metric/code_eval

Method	Pass@k		
	k=1	2	5
WizardCoder 7B	1.21	2.05	3.35
ARCHCODE$_{\text{MIX}}$	**4.24**	**4.24**	**4.24**

Table 10: Experimental results using *WizardCoder* 7B w/o ICL for code generation on CodeContests. 'ARCHCODE$_{\text{MIX}}$' indicates that code filtering is applied with test cases generated by ARCHCODE using GPT-3.5-Turbo.

Regarding the alignment of sub-requirements to the corresponding test cases for code filtering, we generate all test cases for all sub-requirements in one iteration to minimize LLM calls, as shown in Tables 32 and 34. This approach leverages formatted in-context examples from Tables 24 and 26. Subsequently, test cases are parsed and categorized according to corresponding sub-requirement types, followed by a test run with generated code snippets for code filtering.

A.1 Open-sourced Backbone Models and Java Language

Open-source LLMs We utilized huggingface's text-generation-inference [9] to parallelize *WizardCoder* 7B on two NVIDIA RTX A6000 48GBs for inference purposes exclusively. It took approximately one hour to experiment with one method on the entire HumanEval benchmark. Consistent to the results on HumanEval shown in Table 8, Table 10 also shows that ARCHCODE significantly contributes to Pass@k scores on CodeContests.

Other Programming Languages For sparse fine-tuning, we followed Ansell et al. (2022) to train 3% of the SantaCoder 1B (Allal et al., 2023) parameters with the batch size of 8 (1*grad_accum of 8), the learning rate of 2e-5, the L1 regularization of 0, and the max training epochs of 3, using a single NVIDIA RTX A6000 48GB for 2 hours. For the training set, we utilized MegaCodeTraining,[10] a public dataset set, while using java related data only.

B HumanEval-NFR Construction

The HumanEval-NFR benchmark, an extension of HumanEval (Chen et al., 2021), evaluates both FRs

[9] https://huggingface.co/docs/text-generation-inference

[10] https://huggingface.co/datasets/rombodawg/MegaCodeTraining

Requirements	Subtype	# GT TCs
from HumanEval		
Functional	General + Edge	8.1
Additionally Annotated		
Functional	General	3.1
	Edge	2.6
Non-Functional	Time Perf.	1.8
	Robustness	2.3
	Maintainability	1.0

Table 11: The average number of ground truth test cases (GT TCs) per problem on HumanEval-NFR. Note that reliability is confirmed by checking whether other test cases completed gracefully (without errors), regardless of whether the output was correct. Regarding the maintainability, one test case was sufficient as we defined it as whether the generated code exhibits the specified level of Cyclomatic Complexity or not.

and NFRs of code. HumanEval-NFR comprises the same 164 problems as in the original HumanEval suite. While encompassing all the problem descriptions and ground truth test cases from the original HumanEval benchmark for FR verification, it introduces additional test cases for FR and NFR verification. The statistics of HumanEval-NFR's ground truth test cases are shown in Table 11.

Writing new ground truth test cases involved a two-step process. First, we generated candidate test cases based on the existing HumanEval problems using ARCHCODE based on GPT-3.5-Turbo. Second, we revised those test cases both in automatic or manual manner to ensure the quality of the test suite, based on the following protocols.

B.1 Quality Control for FR Test Cases

For candidate test cases evaluating FRs, we executed the ground truth code from the original HumanEval benchmark against each test case tailored to functional requirements. Those that the ground truth code does not pass were discarded.

B.2 Quality Control for NFR Test Cases

For candidate test cases verifying NFRs, three authors manually validated the quality of generated test cases. During validation, the authors adhered to the following principles:

- Misclassified test cases should be rectified, and any duplicates should be eliminated.
- Test cases should compatible to the original ground truth code. If any discrepancy is found in the code, or if a test case is deemed impractical or overly complex, adjustments should be made to ensure it aligns with the original problem description.

In addition, the authors consider guidelines specific to each NFR category:

Time Performance As *Rice's Theorem* (Rice, 1953) states, all non-trivial properties of Turing-recognizable languages are undecidable, which in essence means that there could be no 'time-complexity checkers.' Therefore, HumanEval-NFR follows conventional strategies used in competitive programming contests, such as Codeforces,[11] where code is executed with relatively larger inputs to ensure that inefficient implementations cannot complete within the specified timeout. Specifically, we set the timeout as 5 seconds for all problems.

Robustness Test cases for this category verify whether the implementation gracefully handles diverse types of invalid inputs, such as a string passed as an argument where an integer is expected. For technical reasons, we expect the code to return values like `None`, an empty list, or `False`—all of which are logically evaluated as False in the Python language—rather than forcing it to raise exceptions or using any other means to indicate it has detected an abnormal input.

Maintainability To validate maintainability, we consider code complexity, which affects the ease of understanding and updating the code (Magel et al., 1982). Specifically, HumanEval-NFR computes the Cyclomatic Complexity (CC; McCabe, 1976) of code, which evaluates code complexity by accounting for the depth of nested indented blocks, then checks whether the observed CC score is lower than the threshold. The threshold is set to 5 if the ground truth code from the original HumanEval benchmark has a CC value below 5; if the CC value exceeds 5, we set the threshold as 10 (Watson et al., 1996).

Reliability Rather than generating dedicated test cases, HumanEval-NFR assesses code reliability by executing all the ground truth test cases for the problem and checks if any runtime errors are raised, without verifying if the outputs are correct. This approach aligns with the category's focus on minimizing system failures and extending the mean-time-to-failure.

[11] https://codeforces.com

	All		Time Perf.		Robustness		Maintainability		Reliability	
Pass@k	k=1	5	k=1	5	k=1	5	k=1	5	k=1	5
GPT-3.5-Turbo	2.62	10.03	53.48	65.75	4.21	14.55	53.23	**68.38**	20.98	36.72
+ NFR Instruction	10.70	26.30	48.23	63.22	14.02	34.65	43.54	60.73	62.07	90.79
GPT-3.5-Turbo + CoT	5.00	12.08	50.00	66.03	7.32	17.67	44.33	62.00	45.49	66.83
+ NFR Instruction	5.30	17.50	50.00	66.03	7.62	24.04	43.66	61.87	65.55	93.64
ARCHCODE − Filtering	15.85	24.83	51.52	65.87	25.24	38.82	44.15	59.41	54.33	70.39
+ NFR Instruction	19.80	30.20	51.34	65.24	28.66	43.37	44.33	57.69	88.48	**95.56**
ARCHCODE	25.19	27.33	62.86	**69.70**	40.86	42.72	**56.43**	62.23	68.53	74.67
+ NFR Instruction	**29.50**	**32.88**	**62.99**	67.10	**43.46**	**47.13**	54.21	59.42	**92.46**	95.01

Table 12: Experimental results of requirements instruction prompting on HumanEval-NFR. Boldfaced and underlined values indicate the 1st and 2nd largest scores, respectively. '+ NFR Instruction' means that the further prompt engineered instruction for NFR consideration shown in Table 13 is applied. '− Filtering' denotes an ablated version of ARCHCODE, without code filtering.

```
# Code must satisfy not only functional requirements but also the following non-functional requirements.
# Non-functional Requirements
## Performance: Pertains to time-centric aspects such as algorithmic time complexity or stipulated timeout conditions.
## Robustness: Ensures that code is resilient to invalid inputs.
## Maintainability: Considers factors that contribute to the ease of maintenance.
## Reliability: Ensures that code can handle errors gracefully without causing system failures over an extended period.
Write a code for the problem.
```

Table 13: Engineered prompt which further specified the details of each NFR, used in Table 12.

C Gains from Prompt Engineering

In this study, we did not focus on devising sophisticated prompts, as our main contribution does not rely heavily on using prompt-engineered instructions. Therefore, we can expect even more performance gains when the prompt is further engineered as in Table 13, as we intentionally kept prompt simple in our main experiments.

Table 12 shows that ARCHCODE is scalable to requirement instruction prompts, showing the best performance on HumanEval-NFR (All) when both are applied. Unlike CoT and NFR Instruction that improve Robustness and Reliability only, ARCHCODE contributes to all NFR types. Notably, time performance and maintainability are enhanced solely by ARCHCODE's code filtering, highlighting the unique contribution over prompt engineering.

D Correctness of Generated Requirements

Format As presented in Figure 6, we organized the structure of software requirements into two parts: problem-agnostic and problem-specific. The former describes general guidelines throughout problems related to reliability, performance, and maintainability. The latter includes more specific instructions depending on the problem description, including all three subtypes of functional requirements, the target time complexity for time performance, the invalid conditions for robustness, and the target Cyclomatic Complexity for maintainability.

Validation To confirm the correctness of the requirements generated by ARCHCODE with GPT-3.5-Turbo, we randomly selected three problems, one for each of the following categories: (1) all, (2) some or (3) none of the generated code samples passed the tests. We manually verified validity of each set of requirements for each case, of which the results are summarized in Table 17. Surprisingly, all the generated requirements were correct, regardless of the corresponding generated code's correctness.

E Analysis of Generated Test Cases by ARCHCODE

Table 14 shows the average number of generated test cases by ARCHCODE for each requirement category. Table 15 reports the accuracy of gener-

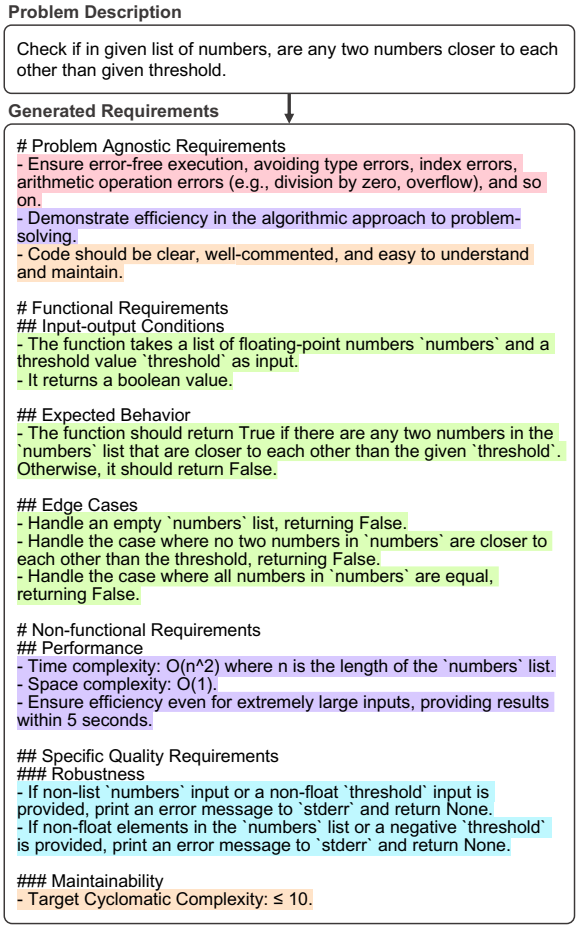


Figure 6: A real-life example of generated requirements from `HumanEval-NFR/0` by ARCHCODE. Best viewed in color.

Requirements	Subtype	# gen. TCs
Functional	General	3.1
	Edge	3.1
Non-Functional	Time Perf.	1.9
	Robustness	2.0
	Maintainability	1.0

Table 14: The average number of generated test cases by ARCHCODE per problem on HumanEval-NFR. Note that reliability is confirmed by checking whether other test cases completed gracefully (without errors), regardless of whether the output was correct. Regarding the maintainability, one test case was sufficient as we defined it as whether the generated code exhibits the specified level of Cyclomatic Complexity or not.

Method	Subtype	Acc.
HumanEval		
ARCHCODE	General	89%
	Edge	49%
CodeContests		
ARCHCODE	General	64%
	Edge	18%

Table 15: The pass ratio of the ground-truth code against generated test cases tailored to functional requirements on HumanEval.

ated test cases tailored to functional requirements. Although the accuracy of generated edge cases is relatively low, they still play a key role in code filtering as evidentiated by the performance discrepancy between ARCHCODE and CODET (w/o clustering) presented in Tables 4. It is noteworthy that CODET generates 'general' test cases, and the results for ARCHCODE and CODET are comparable, given that both methods are based on the same GPT-3.5-Turbo architecture. We conjecture that the relatively low accuracy of generated edge cases does not prevent them from being substantially useful in code filtering, for that wrongful test cases tend to accept or to reject both the correct and incorrect code, rather than selectively passing incorrect ones. In other words, the overall ranking of the generated code samples is hardly affected by the wrongly generated test cases.

For the validation of non-functional requirements, we can implicitly confirm through Figure 5 as using targeted test cases (filled green) yielded better results than using all test cases (empty green) across all NFR categories, as mentioned in Section 5.3.

F NFR Preference Control

Unlike for FRs, one can consider adopting preferences among NFRs, as (1) they inherently describe rather 'optional' tweaks that can additionally guide the behavior of the code and (2) some trade-off relationships among different NFRs (Chung et al., 2012; Krummenacher et al., 2007; Moreira et al., 2005; Gross and Yu, 2001). An example of the latter would be the trade-off between the time performance and the rest of the NFRs (Krummenacher et al., 2007; Gross and Yu, 2001). We have already shown in Section 5.3 that such control can be achieved by adjusting the weights of test cases for different NFR categories in the adjusting approach.

Here, we present an alternative means of accomodating preference, by guiding the generation of code and test cases in a preference-aware manner as with the following two methods:

- **Preference Control by Instruction**: We in-

NFR(s) shown in few-shot examples	Preferred NFR(s)	All		Time Perf.		Robustness		Maintainability		Reliability	
		k=1	5	k=1	5	k=1	5	k=1	5	k=1	5
No Preference											
All	*None*	15.85	24.83	51.52	65.87	25.24	38.82	44.15	59.41	54.33	70.39
Preference Control by Instruction											
All	*Time Perf.*	15.79	26.50	52.38	**68.96**	25.30	40.22	44.15	63.40	**86.10**	**95.18**
All	*All - Time Perf.*	15.55	26.75	51.28	65.56	25.61	41.02	45.55	60.94	82.44	94.13
Preference Control by Plug-and-Play											
Time Perf.	*Time Perf.*	3.84	7.85	**53.29**	67.55	7.93	15.45	**49.45**	**66.65**	35.98	56.24
All - Time Perf.	*All - Time Perf.*	**17.50**	**28.74**	52.01	66.69	**27.68**	**43.63**	44.15	60.22	49.82	68.13

Table 16: NFR preference control of ARCHCODE without applying code filtering (i.e. ARCHCODE − Filtering). Boldfaced and underlined values indicate the 1st and 2nd largest scores, respectively. *All* means all NFRs—time performance, robustness, maintainability, and reliability—are targeted, and *All - Time Perf.* means all NFRs except for time performance are targeted. In the 'Preference Control by Instruction' setting, all NFRs are included in the prompt, and an additional instruction to prioritize specific NFR(s) is appended. In the 'Plug-and-Play' setting, only targeted NFRs are included in the few-shot examples, while no preference among them is assumed.

clude all NFRs in prompts just as before, while explicitly expressing the preference at the end (e.g. *"Consider the time performance requirement to be the most important."*).

- **Preference Control by Plug-and-Play**: We only present the preferred NFRs in prompts, without an explicit description of the preference. All included NFRs are considered equally important, with no prioritization among them.

Table 16 shows that the plug-and-play approach inflicts a larger impact on Pass@k on HumanEval-NFR (All), compared to the instruction-based method. Notably, the plug-and-play approach considering all but time performance showed the best Pass@k scores, which we attribute to the trade-off between time performance and the rest of the NFRs: focusing on the other requirements is relatively free of negative interference that would hurt the performance. In the categories of time performance and robustness, both instruction and plug-and-play preference settings showed improvements when each of them was targeted. For the maintainability category, the best results were observed when only time performance was preferred in the plug-and-play approach. This is likely due to the omission of code lines for handling exceptions related to invalid inputs, as the robustness category was not considered. In the case of reliability, which is assessed by the error-free execution of code across all test cases (without dedicated test cases), performance improvement was observed irrespective of preference in the instruction-based approach. As demonstrated in Table 12, this suggests that prompt engineering can reduce error ratios; we leave exploration towards this direction to future work.

G Varying Difficulty Levels of NFRs in HumanEval-NFR

Orthogonal to ARCHCODE's contribution towards satisfying every requirement category, we observe a general trend of relatively low performance in the robustness category compared to others in HumanEval-NFR, as shown in Table 2 and Figure 5. One conjecture is that the difficulty lies among the NFRs as inferred from the original HumanEval benchmark. As the ground truth code snippets are generally not complex, handling large input size (time performance), managing small Cyclomatic Complexity (maintainability), and avoiding runtime error while running other test cases (reliability) might be easier than correctly handling every possible invalid input (robustness).

Index	Generated Requirements (manually validated)	Generated Code (validated by Ground Truth Test Cases)
		HumanEval-NFR/51
0	Correct	Passed
1	Correct	Passed
2	Correct	Passed
3	Correct	Passed
4	Correct	Passed
5	Correct	Passed
6	Correct	Passed
7	Correct	Passed
8	Correct	Passed
9	Correct	Passed
		HumanEval-NFR/11
0	Correct	failed: invalid literal for int() with base 2: ‘’
1	Correct	failed: Failed to handle input strings of different lengths.
2	Correct	failed: Failed to handle large input size.
3	Correct	failed: Failed to handle input strings of different lengths.
4	Correct	Passed
5	Correct	failed: Failed to have a Cyclomatic Complexity less than or equal to 10 by Radon.
6	Correct	failed: Failed to handle input strings of different lengths.
7	Correct	Passed
8	Correct	failed: invalid literal for int() with base 2: ‘’
9	Correct	Passed
		HumanEval-NFR/0
0	Correct	failed: Failed to handle large input size.
1	Correct	failed: Failed to handle large input size.
2	Correct	failed: Failed to handle large input size.
3	Correct	failed: Failed to handle large input size.
4	Correct	failed: Failed to handle large input size.
5	Correct	failed: Failed to handle large input size.
6	Correct	failed: Failed to handle large input size.
7	Correct	failed: Failed to handle large input size.
8	Correct	failed: Failed to handle large input size.
9	Correct	failed: Failed to handle large input size.

Table 17: Validation of generated requirements by ARCHCODE on HumanEval-NFR. Note that each generated code snippet is uniquely tailored to correspond with a distinct set of requirements, ensuring that no code snippet shares the same set of requirements.

Sub-requirement	Generated Requirement	Human Validation	Explanation
HumanEval-NFR/51			
Description: *remove_vowels is a function that takes string and returns string without vowels.*			
General	The function takes a string 'text' as input.	Correct	The function takes string with or without vowels as input.
General	It returns a string.	Correct	The function should return a string without vowels.
General	The function should remove all vowels from the input 'text' and return the modified string.	Correct	The function should return a string without vowels from the input string.
Edge	Handle an empty 'text' string, returning an empty string.	Correct	Input can be empty string. In this case, the function should return empty string.
Edge	Handle the case where 'text' contains only vowels, returning an empty string.	Correct	Input can contain only vowels. In this case, all of vowels should be removed and the function should return empty string.
Edge	Handle the case where 'text' contains no vowels, returning the original string.	Correct	Input can contain no vowels. In this case, there are nothing to be removed and the function should return original string.
Edge	Handle both lowercase and uppercase vowels.	Correct	Input can be string in lowercase and uppercase. Both of them should be handled.
Performance	Time complexity: O(n) where n is the length of the 'text' string.	Correct	To traverse the string once and check whether each character is a vowel or not, and then add non-vowel characters to the result string, it requires $O(n)$ time, where n is the length of the string.
Performance	Space complexity: $O(n)$ where n is the length of the 'text' string.	Correct	Creating a new string to store the non-vowel characters requires additional space. If the length of the input string is n, the length of the resulting string can be at most n, leading to an $O(n)$ space complexity.
Performance	Ensure efficiency even for extremely large inputs, providing results within 5 seconds.	Correct	The inputs should be handled in predetermined testing time.
Robustness	If a non-string input is provided, print an error message to 'stderr' and return None.	Correct	Invalid inputs should be handled and return None.
Maintainability	Target Cyclomatic Complexity: ≤ 5.	Correct	5 indicates low risk, where the code mainly consists of simple blocks.

Table 18: Detailed explanation for the human validation of generated requirements by ARCHCODE (Index: 0 in Table 17) on HumanEval-NFR/51.

Sub-requirement	Generated Requirement	Human Validation	Explanation
HumanEval-NFR/11			
Description: *Input are two strings a and b consisting only of 1s and 0s. Perform binary XOR on these inputs and return result also as a string.*			
General	The function takes two strings 'a' and 'b' as input.	Correct	The function should take two strings consisting only of 1s and 0s.
General	It returns a string.	Correct	The function should return string having result of a binary XOR operation on these inputs.
General	The function should perform a binary XOR operation on the input strings 'a' and 'b' and return the result as a string.	Correct	The function takes two binary strings and return a result of a binary XOR operation.
Edge	Handle empty strings as input, returning an empty string.	Correct	The function should handle empty strings and return empty string.
Edge	Handle strings with different lengths, returning None.	Correct	When the two inputs have different length, a binary XOR operation is impossible.
Edge	Handle strings with characters other than '0' and '1', returning None.	Correct	When the two inputs are not valid binary string, a binary XOR operation is impossible.
Performance	Time complexity: $O(n)$ where n is the length of the longer input string.	Correct	The time complexity of performing binary XOR on two strings is proportional to the length of the strings.
Performance	Space complexity: $O(n)$ where n is the length of the longer input string.	Correct	If a new string is created to store the result, its length would be equal to the length of the input strings.
Performance	Ensure efficiency even for extremely large inputs, providing results within 5 seconds.	Correct	The inputs should be handled in predetermined testing time.
Robustness	If non-string inputs are provided, print an error message to 'stderr' and return None.	Correct	Invalid inputs should be handled and return None.
Robustness	If the input strings contain characters other than '0' and '1', print an error message to 'stderr' and return None.	Correct	Invalid inputs should be handled and return None.
Maintainability	Target Cyclomatic Complexity: ≤ 10.	Correct	10 indicates low risk, where the code consists of well-structured and stable blocks.

Table 19: Detailed explanation for the human validation of generated requirements by ARCHCODE (Index: 0 in Table 17) on HumanEval-NFR/11.

Sub-requirement	Generated Requirement	Human Validation	Explanation
HumanEval-NFR/0			
Description: *Check if in given list of numbers, are any two numbers closer to each other than given threshold.*			
General	The function takes a list of floating-point numbers 'numbers' and a threshold value 'threshold' as input.	Correct	The function takes two inputs. The first is a list of floating-point numbers and the second is a threshold value.
General	It returns a boolean value.	Correct	The function should return True if there are any two numbers in the 'numbers' list that are closer to each other than the given 'threshold'. Otherwise, it should return False.
General	The function should return 'True' if there are any two numbers in the 'numbers' list that are closer to each other than the given 'threshold'. Otherwise, it should return 'False'.	Correct	The function checks if there are any two numbers closer to each other than given threshold in given list of numbers.
Edge	Handle an empty 'numbers' list, returning 'False'.	Correct	The function should return False if there are not any numbers in the 'numbers' list.
Edge	Handle the case where no two numbers in 'numbers' are closer to each other than the 'threshold', returning 'False'.	Correct	The function should return True if there are any two numbers in the 'numbers' list that are closer to each other than the given 'threshold'. Otherwise, it should return False.
Edge	Handle the case where all numbers in 'numbers' are closer to each other than the 'threshold', returning 'True'.	Correct	The function should return True if there are any two numbers in the 'numbers' list that are closer to each other than the given 'threshold'. Otherwise, it should return False.
Performance	Time complexity: $O(n^2)$ where n is the length of the 'numbers' list.	Correct	All elements of the given list need to be iterated through, and the difference between each pair of numbers needs to be calculated. This process is proportional to the size of the input list. Therefore, if we denote the length of the input list as n, the time complexity is $O(n^2)$.
Performance	Space complexity: $O(1)$.	Correct	If the problem is solved without using additional memory, the space complexity is $O(1)$.
Performance	Ensure efficiency even for extremely large inputs, providing results within 5 seconds.	Correct	The inputs should be handled in predetermined testing time.
Robustness	If non-list 'numbers' input or a non-float 'threshold' input is provided, print an error message to 'stderr' and return None.	Correct	Invalid inputs should be handled and return None.
Robustness	If non-float elements in the 'numbers' list or a negative 'threshold' is provided, print an error message to 'stderr' and return None.	Correct	Invalid inputs should be handled and return None.
Maintainability	Target Cyclomatic Complexity: ≤ 10.	Correct	10 indicates low risk, where the code consists of well-structured and stable blocks.

Table 20: Detailed explanation for the human validation of generated requirements by ARCHCODE (Index: 0 in Table 17) on HumanEval-NFR/0.

Index	Generated Test Cases	Sub-requirement Type	# Passed Cands.	Human Validation
	HumanEval-NFR/51			
0	assert remove_vowels(‘’) == ‘’, ‘Failed to handle an empty input string.’	General	10/10	Correct
1	assert remove_vowels("abcdef\nghijklm") == ‘bcdf\nghjklm’, ‘Failed to remove all vowels from the input string.’	General	10/10	Correct
2	assert remove_vowels(‘abcdef’) == ‘bcdf’, ‘Failed to remove vowels from the input string.’	General	10/10	Correct
3	assert remove_vowels(‘aaaaa’) == ‘’, ‘Failed to handle case where the input string contains only vowels.’	Edge	10/10	Correct
4	assert remove_vowels(‘aaBAA’) == ‘B’, ‘Failed to remove only lowercase vowels from the input string.’	Edge	10/10	Correct
5	assert remove_vowels(‘zbcd’) == ‘zbcd’, ‘Failed to handle case where the input string contains no vowels.’	Edge	10/10	Correct
6	assert remove_vowels(‘a’ * 10**6) == ‘’, ‘Failed to handle large input size.’	Performance	10/10	Correct
7	assert remove_vowels(‘z’ * 10**6) == ‘z’ * 10**6, ‘Failed to handle large input size.’	Performance	10/10	Correct
8	assert remove_vowels(123) == None, ‘Failed to handle case where the input is not a string.’	Robustness	10/10	Correct
9	assert result.total_complexity <= 5, ‘Failed to have a Cyclomatic Complexity less than or equal to 5 by Radon.’	Maintainability	10/10	Correct
	HumanEval-NFR/83			
0	assert starts_one_ends(1) == 9, ‘Failed to count the numbers of 1-digit positive integers that start or end with 1.’	General	0/10	Incorrect
1	assert starts_one_ends(2) == 19, ‘Failed to count the numbers of 2-digit positive integers that start or end with 1.’	General	0/10	Correct
2	assert starts_one_ends(3) == 271, ‘Failed to count the numbers of 3-digit positive integers that start or end with 1.’	General	0/10	Incorrect
3	assert starts_one_ends(0) == 0, ‘Failed to handle the case where n is 0.’	Edge	0/10	Incorrect
4	assert starts_one_ends(-5) == 0, ‘Failed to handle the case where n is negative.’	Edge	0/10	Incorrect
5	assert starts_one_ends(10**6) == 900000, ‘Failed to handle large input size.’	Performance	0/10	Incorrect
6	assert starts_one_ends(‘invalid’) == None, ‘Failed to handle case where the input n is not an integer.’	Robustness	10/10	Correct
7	assert result.total_complexity <= 5, ‘Failed to have a Cyclomatic Complexity less than or equal to 5 by Radon.’	Maintainability	2/10	Correct
	HumanEval-NFR/100			
0	assert make_a_pile(3) == [3, 5, 7], ‘Failed to create the pile correctly.’	General	5/10	Correct
1	assert make_a_pile(5) == [5, 7, 9, 11, 13], ‘Failed to create the pile correctly.’	General	5/10	Correct
2	assert make_a_pile(0) == [], ‘Failed to handle the case where n is 0.’	Edge	5/10	Correct
3	assert make_a_pile(-5) == None, ‘Failed to handle the case where n is negative.’	Edge	5/10	Correct
4	assert make_a_pile(10**6) == list(range(10**6, 10**6 + 2 * 10**6, 2)), ‘Failed to handle large input size.’	Performance	0/10	Correct
5	assert make_a_pile(‘invalid’) == None, ‘Failed to handle the case where the input n is not an integer.’	Robustness	5/10	Correct
6	assert make_a_pile(0.5) == None, ‘Failed to handle the case where the input n is not a positive integer.’	Robustness	5/10	Correct
7	assert result.total_complexity <= 10, ‘Failed to have a Cyclomatic Complexity less than or equal to 10 by Radon.’	Maintainability	10/10	Correct

Table 21: Validation of generated test cases by ARCHCODE on HumanEval-NFR.

Preferred NFR(s)	
Time Perf.	*All - Time Perf.*
	# Problem Agnostic Requirements - Ensure error-free execution, avoiding type errors, arithmetic operation errors (e.g., division by zero, overflow), and so on. - Demonstrate efficiency in the algorithmic approach to problem-solving. - Code should be clear, well-commented, and easy to understand and maintain.
# Functional Requirements ## Input-output Conditions - The function takes an integer ‘decimal‘ as input. - It returns a string.	# Functional Requirements ## Input-output Conditions - The function takes an integer ‘decimal‘ as input. - It returns a string.
## Expected Behavior - The function should convert the given decimal number to binary and return a string representation of the binary number. - The resulting string should have the characters ‘db’ at the beginning and end.	## Expected Behavior - The function should convert the decimal number to binary and return a string representation of the binary number. - The string should have the format "db" + binary representation + "db". - Each character in the binary representation should be ‘0’ or ‘1’.
## Edge Cases - Handle negative decimal numbers, returning the binary representation with a ‘-’ sign. - Handle decimal numbers equal to 0, returning the string "db0db".	## Edge Cases - Handle the case where ‘decimal‘ is 0, returning "db0db". - Handle the case where ‘decimal‘ is negative, returning None. - Handle the case where ‘decimal‘ is not an integer, returning None.
# Non-functional Requirements ## Performance - Time complexity: O(log n) where n is the value of the ‘decimal‘ input. - Space complexity: O(log n) where n is the value of the ‘decimal‘ input. - Ensure efficiency even for extremely large inputs, providing results within 5 seconds.	# Non-functional Requirements ## Specific Quality Requirements ### Robustness - If a non-integer ‘decimal‘ input is provided, print an error message to ‘stderr‘ and return None. - If a negative ‘decimal‘ input is provided, print an error message to ‘stderr‘ and return None. ### Maintainability - Target Cyclomatic Complexity ≤ 5.

Table 22: Real life examples of generated requirements for `HumanEval-NFR/79` by ARCHCODE with Plug-and-Play NFR Preference Control.

Preferred NFR(s)

Time Perf.

```
import sys
import math
import re
import numpy
import numpy as np
from typing import *

def decimal_to_binary(decimal):
    binary = bin(decimal)[2:] # Convert decimal to binary and remove ‘0b’ prefix
    return ‘db’ + binary + ‘db’ # Add ‘db’ prefix and suffix to the binary representation
```

All - Time Perf.

```
import sys
import math
import re
import numpy
import numpy as np
from typing import *

def decimal_to_binary(decimal):
    # Input Validation

    if not isinstance(decimal, int):

        sys.stderr.write('Invalid input: decimal must be an integer.')
    return None

    # Initialize variables
    binary = ""

    # Handle special case for 0
    if decimal == 0:
        return "db0db"

    # Convert decimal to binary
    while decimal > 0:
        binary += str(decimal % 2)
        decimal //= 2

    # Reverse the binary string
    binary = binary[::-1]

    # Add prefix and suffix
    binary = "db" + binary + "db"

    return binary
```

Table 23: Real life examples of generated code snippets for `HumanEval-NFR/79` by ARCHCODE with Plug-and-Play NFR Preference Control. The conditioned requirements are shown in Table 22.

H In-Context Learning Examples

Problem Description Example

```
from typing import List

def longest_subarray_with_sum_limit(nums: List[int], target: int) -> List[int]:
    """Find the longest subarray of the given array ‘nums’ such that the sum of its elements is less than or equal to the given ‘target’.
        >>> longest_subarray_with_sum_limit([3, 1, 5, 2, 1, 9, 7, 4], 10)
        [3, 1, 5]
        >>> longest_subarray_with_sum_limit([1, 2, 3, 4, 5], 3)
        [1, 2]
    """
```

Requirement Generation Prompt

Write requirements for the problem.

Requirement Generation Example

```
# Problem Agnostic Requirements
- Ensure error-free execution, avoiding type errors, index errors, arithmetic operation errors (e.g., division by zero, overflow), and so on.
- Demonstrate efficiency in the algorithmic approach to problem-solving.
- Code should be clear, well-commented, and easy to understand and maintain.

# Functional Requirements
## Input-output Conditions
- The function takes a list of integers ‘nums’ and a target integer ‘target’ as input.
- It returns a list of integers.

## Expected Behavior
- The function should return the longest subarray of ‘nums’ whose sum is less than or equal to ‘target’.

## Edge Cases
- Handle an empty ‘nums’ list, returning an empty list.
- Handle the case where no subarray in ‘nums’ satisfies the condition, returning an empty list.
- Handle multiple subarrays with the same maximum length, returning any of them.

# Non-functional Requirements
## Performance
- Time complexity: O(n) where n is the length of the ‘nums’ list.
- Space complexity: O(1).
- Ensure efficiency even for extremely large inputs, providing results within 5 seconds.

## Specific Quality Requirements
### Robustness
- If non-list ‘nums’ input or a non-integer ‘target’ input is provided, print an error message to ‘stderr’ and return None.
- If non-integer elements in the ‘nums’ list or a negative ‘target’ is provided, print an error message to ‘stderr’ and return None.

### Reliability
- Avoid index errors while accessing ‘nums’ elements.

### Maintainability
- Target Cyclomatic Complexity ≤ 10.
```

Table 24: The in-context example for requirement generation by ARCHCODE. The problem description example is from Shinn et al. (2023).

Problem Description Example

from typing import List ...

Requirement Generation Prompt

Write requirements for the problem.

Requirement Generation Example

Problem Agnostic Requirements ...

Code Generation Prompt

Write the code for the problem.

Code Generation Example

```
from typing import List, Optional
import sys

def longest_subarray_with_sum_limit(nums: Optional[List[int]], target: Optional[int]) -> List[int]:
    # Validate inputs
    if not isinstance(nums, list) or not all(isinstance(num, int) for num in nums):
        sys.stderr.write('Invalid input: nums must be a list of integers.')
        return
    if not isinstance(target, int):
        sys.stderr.write('Invalid input: target must be an integer.')
        return

    n = len(nums)
    left, right = 0, 0
    max_length = 0
    current_sum = 0
    result = []

    # Handle empty list
    if n == 0:
        return []

    # Initial window expansion
    while right < n and current_sum + nums[right] <= target:
        current_sum += nums[right]
        right += 1

    # Update result if a valid window was found during initial expansion
    if right > 0:
        result = nums[left:right]
        max_length = right

    # Main loop: Slide the window through the array
    while right < n:
        current_sum += nums[right]

        # Shrink window from the left if current_sum exceeds target
        while current_sum > target:
            current_sum -= nums[left]
            left += 1

        # Update result if a longer valid window is found
        if right - left + 1 > max_length:
            max_length = right - left + 1
            result = nums[left:right+1]

        right += 1

    return result
```

Table 25: The in-context example for code generation by ARCHCODE. The skipped contents—Problem Description Example and Requirement Generation Example—are available in Table 24.

Problem Description Example

from typing import List ...

Requirement Generation Prompt

Write requirements for the problem.

Requirement Generation Example

Problem Agnostic Requirements ...

Test Case Generation Prompt

Write test cases for the problem.

Test Case Generation Example

```
# Test Cases Regarding Functional Requirements
## General Cases
### The longest subarray with sum less than or equal to 10 is [1, 5, 2, 1]
assert longest_subarray_with_sum_limit([3, 1, 5, 2, 1, 9, 7, 4], 10) == [1, 5, 2, 1], 'Failed to find the longest subarray.'

### None of the subarrays have a sum less than or equal to 3
### The function should return an empty list
assert longest_subarray_with_sum_limit([2, 3, 4, 5, 6], 1) == [], 'Failed to handle case where no subarray satisfies the condition.'

## Edge Cases
### The input list is empty, so the function should return an empty list
assert longest_subarray_with_sum_limit([], 5) == [], 'Failed to handle an empty input list.'

### The longest subarray with sum less than or equal to 10 is [5, 5]
assert longest_subarray_with_sum_limit([5, 5, 5, 5], 10) == [5, 5], 'Failed to find the longest subarray when all elements have the same value.'

### The entire nums array is a valid subarray with sum less than or equal to 15
assert longest_subarray_with_sum_limit([1, 2, 3, 4, 5], 15) == [1, 2, 3, 4, 5], 'Failed to handle case where the entire array is a valid subarray.'

# Test Cases Regarding Non-functional Requirements
## Performance Requirements
### The nums list contains 10^6 elements with increasing values from 1 to 10^6
### The longest subarray with sum less than or equal to 10^5 is [1, 2, 3, ..., 10^5]
assert longest_subarray_with_sum_limit(list(range(1, 10**6 + 1)), 10**5) == list(range(1, 446 + 1)), 'Failed to handle large input size.'

### The nums list contains 10^6 elements with increasing values from 1 to 10^6
### The longest subarray with sum less than or equal to 2 is [1]
assert longest_subarray_with_sum_limit(list(range(1, 10**6 + 1)), 2) == [1], 'Failed to handle case where the subarray length is 1.'

### The nums list contains 10^6 elements with increasing values from 1 to 10^6
### The longest subarray with sum less than or equal to 5 is [1, 2]
assert longest_subarray_with_sum_limit(list(range(1, 10**6 + 1)), 5) == [1, 2], 'Failed to handle case where the subarray length is 2.'

### The nums list contains 10^6 elements, all of which are 10^6
### The sum of all the elements in nums is 10^12, which is larger than the target
### Therefore, there is no subarray that satisfies the condition and the function should return an empty list
assert longest_subarray_with_sum_limit([10**8] * 10**6, 10**7) == [], 'Failed to handle case where no subarray satisfies the condition.'

### The nums list contains 10^6 elements with increasing values from 1 to 10^6
### The target is the sum of all the elements in nums, so the entire array is a valid subarray
assert longest_subarray_with_sum_limit(list(range(1, 10**5 + 1)), 10**10) == list(range(1, 10**5 + 1)), 'Failed to handle case where the entire array is a valid subarray.'

## Specific Quality Requirements
### Robustness
#### The sum of the elements in nums is larger than the target,
#### so the function should return None
assert longest_subarray_with_sum_limit([10**20, 10**20], 10**19) == None, 'Failed to handle case where the sum of the elements in nums is larger than the target.'

#### The nums input is not a list of integers, so the function should return None
assert longest_subarray_with_sum_limit(`invalid', 10) == None, 'Failed to handle case where the input nums is not a list.'

#### The target input is not an integer, so the function should return None
assert longest_subarray_with_sum_limit([1, 2, 3], 'invalid') == None, 'Failed to handle case where the input target is not an integer.'

#### The nums list contains elements that are not integers, so the function should return None
assert longest_subarray_with_sum_limit([1, 2, 'invalid', 4], 5) == None, 'Failed to handle case where the input nums contains non-integer elements.'

#### The target is a negative number, so the function should return None
assert longest_subarray_with_sum_limit([1, 2, 3, 4, 5], -10) == None, 'Failed to handle case where the input target is negative.'

### Reliability
#### Satisfied if no errors occur across all test cases

### Maintainability
#### Calculate Cyclomatic Complexity using Radon
#### Check if the Cyclomatic Complexity is less than or equal to 10
from radon.visitors import ComplexityVisitor
result = ComplexityVisitor.from_code("""${Generated Code}""")
assert result.total_complexity <= 10, 'Failed to have a Cyclomatic Complexity less than or equal to 10 by Radon.'
```

Table 26: The in-context example for test case generation by ARCHCODE. The skipped contents—Problem Description Example and Requirement Generation Example—are available in Table 24. **${Generated Code}** denotes the string text of the code that is to be checked.

I In-Context Learning Prompt Templates

{Problem Description Example}
Write requirements for the problem.
{Requirement Generation Example}

{Test Problem Description}
Write requirements for the problem.

Table 27: The in-context learning prompt template for requirement generation by ARCHCODE.

{Problem Description Example}
Write requirements for the problem.
{Requirement Generation Example}
Write the code for the problem.
{Code Generation Example}

{Test Problem Description}
Write requirements for the problem.
{Generated Requirements by ARCHCODE}
Write the code for the problem.

Table 28: The in-context learning prompt template for code generation by ARCHCODE.

{Problem Description Example}
Write requirements for the problem.
{Requirement Generation Example}
Write test cases for the problem.
{Test Case Generation Example}

{Test Problem Description}
Write requirements for the problem.
{Generated Requirements by ARCHCODE}
Write test cases for the problem.

Table 29: The in-context learning prompt template for test case generation by ARCHCODE.

{Problem Description Example}
Write the code for the problem.
{Code Generation Example}

{Test Problem Description}
Write the code for the problem.

Table 30: The in-context learning prompt template for code generation by GPT-3.5-Turbo baseline.

{Problem Description Example}
Write the plan for the problem.
{Chain-of-Thought Example}
Write the code for the problem.
{Code Generation Example}

{Test Problem Description}
Write the plan for the problem.

Table 31: The in-context learning prompt template for code generation by GPT-3.5-Turbo + CoT baseline.

J Case Study

Generated Requirements

Problem Agnostic Requirements
- Ensure error-free execution, avoiding type errors, index errors, arithmetic operation errors (e.g., division by zero, overflow), and so on.
- Demonstrate efficiency in the algorithmic approach to problem-solving.
- Code should be clear, well-commented, and easy to understand and maintain.

Functional Requirements
Input-output Conditions
- The function takes two integers ‘x‘ and ‘base‘ as input.
- It returns a string.

Expected Behavior
- The function should convert the number ‘x‘ to the given ‘base‘ and return the string representation of the converted number.

Edge Cases
- Handle the case where ‘x‘ is 0, returning ’0’.
- Handle the case where ‘base‘ is 1, returning a string of ’1’ repeated ‘x‘ times.
- Handle the case where ‘base‘ is less than 2 or greater than 9, returning None.
- Handle the case where ‘x‘ is negative, returning None.

Non-functional Requirements
Performance
- Time complexity: O(log(x)) where x is the value of ‘x‘.
- Space complexity: O(log(x)) to store the result string.
- Ensure efficiency even for extremely large inputs, providing results within 5 seconds.

Specific Quality Requirements
Robustness
- If a non-integer ‘x‘ or ‘base‘ input is provided, print an error message to ‘stderr‘ and return None.
- If a negative ‘base‘ is provided, print an error message to ‘stderr‘ and return None.

Maintainability
- Target Cyclomatic Complexity: ≤ 10.

Table 32: A real-life example of generated requirements for HumanEval-NFR/44 by ARCHCODE.

Generated Code

```
def change_base(x: int, base: int) -> str:
    # Input validation
    if not isinstance(x, int) or not isinstance(base, int):
        sys.stderr.write('Invalid input: x and base must be integers.')
        return None
    if base < 2 or base > 9:
        sys.stderr.write('Invalid input: base must be between 2 and 9 (inclusive).')
        return None
    if x < 0:
        sys.stderr.write('Invalid input: x must be a non-negative integer.')
        return None

    # Handle special case
    if x == 0:
        return '0'

    result = ''
    while x > 0:
        remainder = x % base
        result = str(remainder) + result
        x = x // base

    return result
```

Table 33: A real-life example of generated code for HumanEval-NFR/44 by ARCHCODE.

Generated Test Cases

```
# Test Cases Regarding Functional Requirements
## General Cases
### The number 8 in base 3 is represented as '22'
assert change_base(8, 3) == '22', 'Failed to convert number to the desired base.'

### The number 8 in base 2 is represented as '1000'
assert change_base(8, 2) == '1000', 'Failed to convert number to the desired base.'

### The number 7 in base 2 is represented as '111'
assert change_base(7, 2) == '111', 'Failed to convert number to the desired base.'

## Edge Cases
### The input number is 0, so the function should return '0' in any base
assert change_base(0, 3) == '0', 'Failed to handle case where the input number is 0.'

### The input base is 1, which is not a valid base, so the function should return None
assert change_base(8, 1) is None, 'Failed to handle case where the base is not a valid base.'

### The input number is negative, so the function should return None
assert change_base(-8, 2) is None, 'Failed to handle case where the input number is negative.'

### The input base is greater than 9, which is not a valid base, so the function should return None
assert change_base(8, 10) is None, 'Failed to handle case where the base is not a valid base.'

# Test Cases Regarding Non-functional Requirements
## Performance Requirements
### The input number is 10^6, and the base is 2
### The function should return a string representation of the number in base 2 within a reasonable time frame
assert change_base(10**6, 2) == '11110100001001000000', 'Failed to handle large input size.'

### The input number is 10^6, and the base is 9
### The function should return a string representation of the number in base 9 within a reasonable time frame
assert change_base(10**6, 9) == '1783661', 'Failed to handle large input size.'

### The input number is 10^18, and the base is 2
### The function should return a string representation of the number in base 2 within a reasonable time frame
assert change_base(10**18, 2) == '110111100000101101101011001110100111011001000000000000000000', 'Failed to
handle large input size.'

## Specific Quality Requirements
### Robustness
#### The input number is not an integer, so the function should return None
assert change_base('invalid', 2) is None, 'Failed to handle case where the input number is not an integer.'


#### The input base is not an integer, so the function should return None
assert change_base(8, 'invalid') is None, 'Failed to handle case where the input base is not an integer.'

### Reliability
#### Satisfied if no errors occur across all test cases

### Maintainability
#### Calculate Cyclomatic Complexity using Radon
#### Check if the Cyclomatic Complexity is less than or equal to 10
import inspect
from radon.visitors import ComplexityVisitor
result = ComplexityVisitor.from_code("""${Generated Code}""")
assert result.total_complexity <= 10, 'Failed to have a Cyclomatic Complexity less than or equal to 10 by Radon.'
```

Table 34: A real-life example of generated test cases for `HumanEval-NFR/44` by ARCHCODE. **${Generated Code}** denotes the string text of the code that is to be checked.

Generated Requirements

 # Problem Agnostic Requirements
- Ensure execution is error-free, mitigating type errors, index errors, and arithmetic operation errors (e.g., division by zero, overflow) among others.
- Showcase efficiency in the algorithmic approach to problem-solving.
- Ensure code is clear, well-commented, and both easy to understand and maintain.

 # Functional Requirements

Input-output Conditions

Inputs
- Initial input values: n and m
- Must be positioned in the first line of input
- Adhere to the format: "$\{n\}\{m\}$"
- Integer n range: ($0 \leqslant n < 2000$)
- Integer m range: ($0 \leqslant m < 2000$)
- Subsequent input values: grid denoting the positions of telephone poles
- Each input line format: "$\{a_{i,1}\}\{a_{i,2}\}...\{a_{i,m+1}\}$"
- Integer $a_{i,j}$ range: ($0 \leqslant a_{i,j} \leqslant 1$)
- Each line represents a row in the grid
- There is at least one telephone pole in the given grid.

Outputs
- Output must be a single integer denoting the value of $\sum_{x=0}^{n}\{\sum_{y=0}^{m}\{S(x,y)\}\}$

Expected Behavior
- The city is represented as a plane.
- The plane is represented by a grid of size $(n+1) * (m+1)$.
- Each point (x, y) on the plane has a square of the Euclidean distance between the nearest pole and (x, y) denoted as $S(x, y)$.
- $S(x, y)$ is defined as $(x_2 - x_1)^2 + (y_2 - y_1)^2$, where (x_1, y_1) is the nearest pole and (x_2, y_2) is (x, y).
- The code must calculate the sum of all $S(x, y)$ for each $0 \leqslant x \leqslant n$ and $0 \leqslant y \leqslant m$.
- The code must output the value of $\sum_{x=0}^{n}\{\sum_{y=0}^{m}\{S(x,y)\}\}$ as a single integer.

Edge Cases

Ensure the code functions accurately, even if:
- Inputs reach the condition boundaries
- n or m is 0
- n or m is the maximum value
- There is only one telephone pole in the grid

 # Non-functional Requirements

Performance
- Test time limit: 2 seconds
- Test memory limit: 256 megabytes

Specific Quality Requirements

Robustness

Ensure the code operates correctly even if:
- Input contains redundant spaces or newlines

Enforce the code to print an appropriate error message if:
- Input format is incorrect
- Input data type is invalid
- Input is out of the specified range
- The grid does not contain at least one telephone pole.

Reliability
- Mitigate type errors during input and output stream management.
- Prevent index errors while managing grid values.

Maintainability
- Aim for a Cyclomatic Complexity of: $\leqslant 10$.

Table 35: A real-life example of generated requirements for `CodeContests/2` (`1575_M. Managing Telephone Poles`) by ARCHCODE.

Generated Code

```
import sys
import math

def read_line():
    input_str = ""
    while not input_str:
        input_str = input().strip()
    return input_str

line = read_line()
n, m = map(int, line.split())
if not (0 <= n < 2000 and 0 <= m < 2000):
    sys.stderr.write("error at the boundary of n or m")
    exit()

grid = [[0] * (m + 1) for _ in range(n + 1)]
for i in range(n + 1):
    line = read_line()
    if len(line) != m + 1:
        sys.stderr.write("error at the boundary of the grid")
        exit()
    for j in range(m + 1):
        if not (0 <= int(line[j]) <= 1):
            sys.stderr.write("error at the boundary of the grid")
            exit()
        grid[i][j] = int(line[j])

result = 0
for i in range(n + 1):
    for j in range(m + 1):
        if grid[i][j] == 1:
            continue
        minDistance = math.inf
        for x in range(n + 1):
            for y in range(m + 1):
                if grid[x][y] == 1:
                    distance = (x - i) ** 2 + (y - j) ** 2
                    minDistance = min(minDistance, distance)
        result += minDistance

print(result)
```

Table 36: A real-life example of generated code for `CodeContests/2 (1575_M. Managing Telephone Poles)` by ARCHCODE.

Generated Test Cases

```
# Test Cases Regarding Functional Requirements
## General Cases
ERROR_MESSAGE = "general case error"
INPUT = "2 2\n101\n000\n000"
OUTPUT = "18"

ERROR_MESSAGE = "general case error"
INPUT = "5 4\n10010\n00000\n01000\n00001\n00100\n00010"
OUTPUT = "36"
## Edge Cases
ERROR_MESSAGE = "error at the lower boundary of n"
INPUT = "0 2\n00\n00\n00"
OUTPUT = "0"

ERROR_MESSAGE = "error at the upper boundary of n"
INPUT = "1999 2\n" + "0" * 2000 + "\n" + "0" * 2000
OUTPUT = "0"

ERROR_MESSAGE = "error at the lower boundary of m"
INPUT = "2 0\n0\n0\n0"
OUTPUT = "0"

ERROR_MESSAGE = "error at the upper boundary of m"
INPUT = "2 1999\n" + "0\n" * 2000
OUTPUT = "0"

ERROR_MESSAGE = "error when a_{i, j} is not in the correct format"
INPUT = "2 2\n101\n000\n00"
STD_ERROR = "a_{i, j} is not in the correct format"

ERROR_MESSAGE = "error when a_{i, j} is not in the correct range"
INPUT = "2 2\n101\n000\n002"
STD_ERROR = "a_{i, j} is not in the correct range"

ERROR_MESSAGE = "error when there is no telephone pole in the given grid"
INPUT = "2 2\n000\n000\n000"
STD_ERROR = "there is no telephone pole in the given grid"

# Test Cases Regarding Non-functional Requirements
## Performance Requirements
ERROR_MESSAGE = "execution failed while running exceptionally large input"
INPUT = "1999 1999\n" + "1" * 2000 + "\n" * 1999 + "1" * 2000
OUTPUT = "0"
## Specific Quality Requirements
### Robustness
ERROR_MESSAGE = "error when redundant spaces or newlines are added to the input"
INPUT = "2 2\n101\n000\n\n\n"
OUTPUT = "18"

ERROR_MESSAGE = "error when the input is not in the correct format"
INPUT = "2 2\n101\n000\n00"
STD_ERROR = "the input is not in the correct format"

ERROR_MESSAGE = "error when the data type of the input is not correct"
INPUT = "2 2\n101\n000\n00a"
STD_ERROR = "the data type of the input is not correct"

ERROR_MESSAGE = "error when the input is not in the correct range"
INPUT = "2 2\n101\n000\n003"
STD_ERROR = "the input is not in the correct range"

ERROR_MESSAGE = "error when there is no telephone pole in the given grid"
INPUT = "2 2\n000\n000\n000"
STD_ERROR = "there is no telephone pole in the given grid"
## Reliability
Satisfied if no errors occur across all test cases
## Maintainability
ERROR_MESSAGE = "error when cyclomatic complexity is more than the limit"
COMPLEXITY_LIMIT = 10
```

Table 37: A real-life example of generated test cases for `CodeContests/2/1575_M. Managing Telephone Poles` by ARCHCODE. **${Generated Code}** denotes the string text of the code that is to be checked.