NoFunEval: Funny How Code LMs Falter on Requirements Beyond Functional Correctness

Manav Singhal [1] **Tushar Aggarwal** [*] [1] **Abhijeet Awasthi** [*] [1] **Nagarajan Natarajan** [1] **Aditya Kanade** [1]

Abstract

Existing evaluation benchmarks of language models of code (code LMs) focus almost exclusively on whether the LMs can generate functionally-correct code. In real-world software engineering, developers think beyond functional correctness. They have requirements on "how" a functionality should be implemented to meet overall system design objectives like efficiency, security, and maintainability. They would also trust the code LMs more if the LMs demonstrate robust understanding of requirements and code semantics.

We propose a new benchmark NoFunEval to evaluate code LMs on *non-fun*ctional requirements and simple classification instances for both functional and non-functional requirements. We propose a prompting method, *Coding Concepts* (*CoCo*), as a way for a developer to communicate the domain knowledge to the LMs. We conduct an extensive evaluation of twenty-two code LMs. Our finding is that they generally falter when tested on our benchmark, hinting at fundamental blindspots in their training setups. Surprisingly, even the classification accuracy on functional-correctness instances derived from the popular HumanEval benchmark is low, calling in question the depth of their comprehension and the source of their success in generating functionally-correct code in the first place. We will release our benchmark and evaluation scripts publicly at `https://aka.ms/NoFunEval`.

1. Introduction

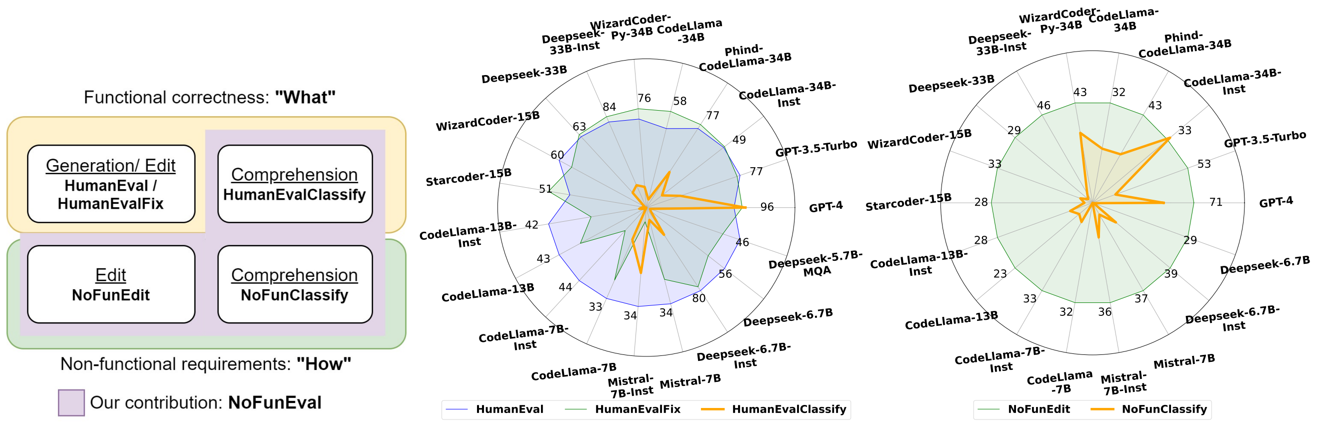
There has been dazzling progress in the development of newer and more capable language models (LMs) of code, e.g., (Chen et al., 2021; Wang & Komatsuzaki, 2021; Austin et al., 2021; Black et al., 2022; Tunstall et al., 2022; Fried et al., 2022; Nijkamp et al., 2023; Li et al., 2023b; Wang et al., 2023; OpenAI, 2023; Luo et al., 2023; Muennighoff et al., 2023). Simultaneously, the community has been actively designing benchmarks (Hendrycks et al., 2021; Chen et al., 2021; Austin et al., 2021; Puri et al., 2021; Li et al., 2022; Liu et al., 2023b) with emphasis on *generating* code for a given problem specification of *what* functionality to achieve, e.g., writing a Python function to sort an array.

This is but a narrow slice of application of LMs in software engineering pipelines where the tasks are often not as straight-forward. Developers must consider the overall requirements of the system (e.g., an Android application) to which the code belongs. So, a problem instance in the real-world would be closer to editing Java code to *optimize for resource usage* on a low-memory Android device than to generating a functionally-correct sort. Such *non-functional requirements* guide the design decisions and constrain *how* the functionality may be realized (Landes & Studer, 1995), and play a central role in real-world software engineering (Chung et al., 2012). The premise of our work is that while satisfying functional requirements ("what" to implement) is necessary, it is not sufficient.

In this work, we forefront the above-identified significant gap in the current evaluation suites, and introduce a complementary benchmark in a first attempt to bridge the gap. We identify a set of *five broad non-functional requirements*: latency, resource utilization, runtime efficiency, maintainability, and security. We design the NoFunEval benchmark consisting of 958 problem instances in multiple programming languages sourced from public repositories and existing datasets, spanning the five non-functional requirements.

As LMs are finding increasing use in code generation and editing, it is imperative to test whether they have *robust comprehension of the requirements and code semantics*. We therefore propose classification instances for both functional-correctness and non-functional requirements, where the former are derived from the HumanEvalFix dataset (Muennighoff et al., 2023). Figure 1(a) shows the three distinct subsets, NoFunEdit, NoFunClassify and HumanEvalClassify, of our NoFunEval benchmark and how

[*]Equal contribution [1]Microsoft Research, India. Correspondence to: Nagarajan Natarajan <nagarajan.natarajan@microsoft.com>, Aditya Kanade <kanadeaditya@microsoft.com>.



(a) Composition of NoFunEval (b) Functional correctness (c) Non-functional requirements

Figure 1: **(a)** NoFunEval contributes edit and comprehension tasks, NoFunEdit and NoFunClassify, for non-functional requirements, and complements HumanEval and HumanEvalFix with a comprehension task HumanEvalClassify. **(b)–(c)**: Comparative model performance on NoFunEval, HumanEval and HumanEvalFix benchmarks. Due to the binary nature of classification instances, we consider those instances of NoFunEdit in plot (c) for which we have a binary-oracle for evaluating edits. For economy of space, we plot only a subset of models. Full results are presented later in the paper.

they complement existing generation and edit benchmarks HumanEval (Chen et al., 2021) and HumanEvalFix (Muennighoff et al., 2023) focused on functional correctness.

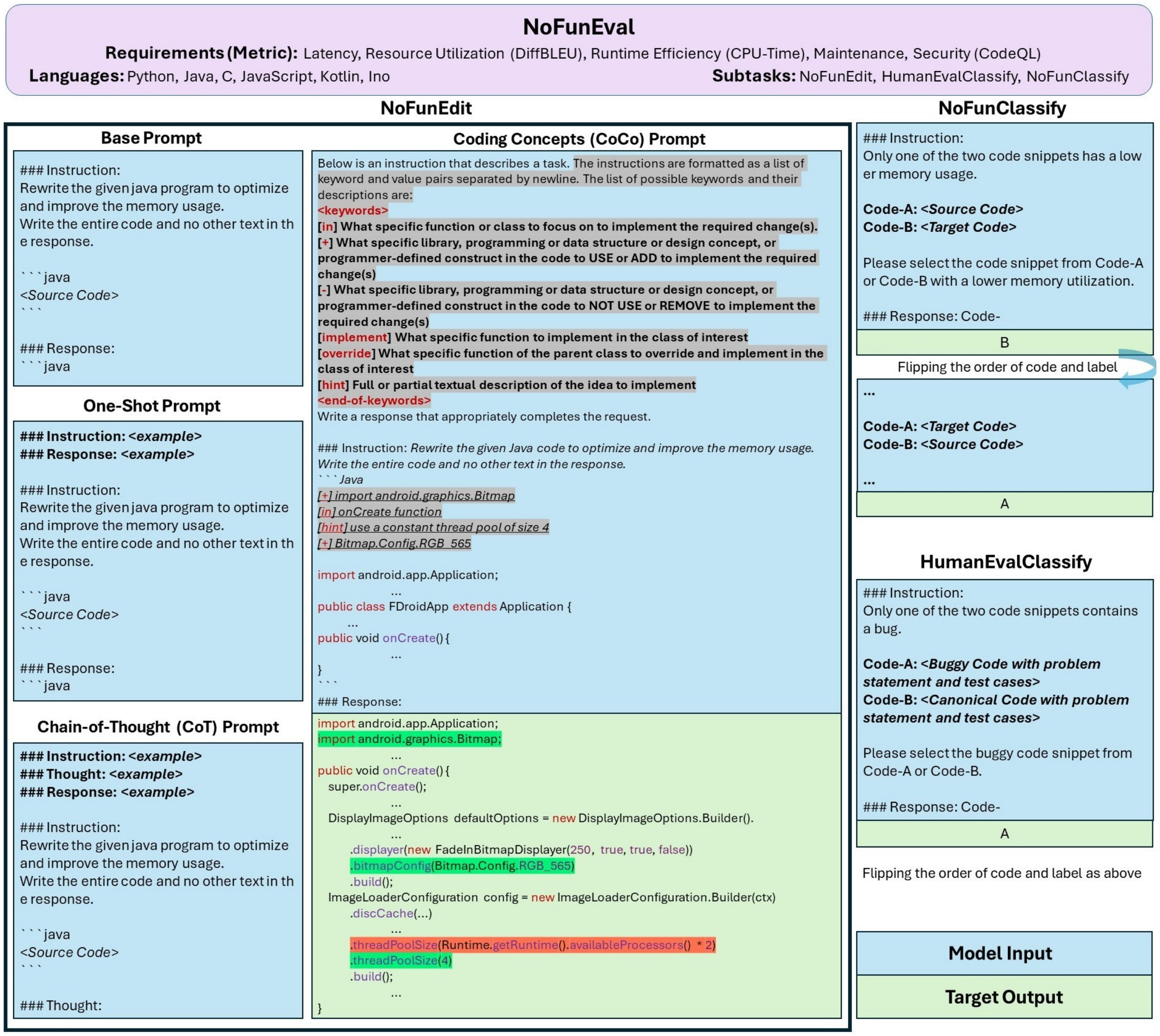
Two key challenges in our benchmark design are: (1) *how do we convey* the notions of latency, security, efficiency that tend to be *relative* unlike functional correctness that is *absolute*? There is no clear prompting strategy known for eliciting general non-functional requirements. Simply giving a description of the requirement in the prompt often fails because LMs lack the necessary domain knowledge, unlike in the case of functional correctness where the problem description is usually sufficient. To this end, we design a new prompting strategy *Coding Concepts* (*CoCo*) which allows a developer to succinctly communicate actionable domain knowledge to the LMs; (2) *how do we evaluate* the output of LMs? We employ a combination of functional (input-output specification) and non-functional (static analysis tools, execution time) oracles. In addition, we consider a BLEU-based metric (Bairi et al., 2023) that operates on the code diffs (diff between source and generated/target code) to capture the closeness of an edit to the ground-truth edit.

We present a comprehensive evaluation of twenty-two code LMs. A key takeaway of our work, besides the benchmark itself, is that existing code LMs (spanning different training strategies, instruction tuning paradigms and model sizes) falter when we test them on requirements beyond functional correctness. This is highlighted in Figures 1(b)–1(c) by (1) their *surprisingly low performance on classification* versus generation and editing tasks in both functional and non-functional requirements, and (2) the *generally low performance on non-functional requirements* compared to higher performances on generation and editing tasks on functional correctness. For instance, the top two open-source models DeepSeekCoder-33B-Inst and Phind-CodeLlama-34B solve 79% and 73% problems in HumanEval; 81% and 77% problems in HumanEvalFix, respectively. However, they get low accuracies of 21% and 34% on the corresponding classification dataset HumanEvalClassify (§ 2.4). On NoFunEdit (§ 2.1), only 38% and 34% solutions generated by them are accepted by the oracles. We believe that these observations hint at fundamental blindspots in the training setups of code LMs that may hinder their practical use going forward.

Our contributions are as follows: We (1) identify the almost exclusive focus on functional correctness in existing evaluation benchmarks of code LMs; (2) prepare a benchmark to evaluate non-functional requirements and comprehension ability of code LMs; (3) conduct an extensive evaluation of twenty-two code LMs finding that there is much room to improve comprehension of requirements and code semantics; and (4) release our benchmark and evaluation scripts publicly at https://aka.ms/NoFunEval.

2. The NoFunEval Benchmark

In Figure 2, we provide an overview of our NoFunEval benchmark, which comprises three types of tasks – (1) NoFunEdit, the task of editing a given code snippet based on a non-functional requirement; (2) NoFunClassify, the task of distinguishing between given code snippets based on a non-functional requirement; (3) HumanEvalClassify, the task of distinguishing between given code snippets based on their functional correctness. NoFunClassify and NoFunEdit are designed to evaluate an LM's understanding of



NoFunEval

Requirements (Metric): Latency, Resource Utilization (DiffBLEU), Runtime Efficiency (CPU-Time), Maintenance, Security (CodeQL)

Languages: Python, Java, C, JavaScript, Kotlin, Ino

Subtasks: NoFunEdit, HumanEvalClassify, NoFunClassify

NoFunEdit

Base Prompt

~~~
### Instruction:
Rewrite the given java program to optimize
and improve the memory usage.
Write the entire code and no other text in th
e response.

```java
<Source Code>
~~~

Response:
```java
```

One-Shot Prompt

~~~
### Instruction: <example>
### Response: <example>

### Instruction:
Rewrite the given java program to optimize
and improve the memory usage.
Write the entire code and no other text in th
e response.

```java
<Source Code>
~~~

Response:
```java
```

Chain-of-Thought (CoT) Prompt

~~~
### Instruction: <example>
### Thought: <example>
### Response: <example>

### Instruction:
Rewrite the given java program to optimize
and improve the memory usage.
Write the entire code and no other text in th
e response.

```java
<Source Code>
```

### Thought:
~~~

Coding Concepts (CoCo) Prompt

Below is an instruction that describes a task. The instructions are formatted as a list of keyword and value pairs separated by newline. The list of possible keywords and their descriptions are:
<keywords>
[in] What specific function or class to focus on to implement the required change(s).
[+] What specific library, programming or data structure or design concept, or programmer-defined construct in the code to USE or ADD to implement the required change(s)
[-] What specific library, programming or data structure or design concept, or programmer-defined construct in the code to NOT USE or REMOVE to implement the required change(s)
[implement] What specific function to implement in the class of interest
[override] What specific function of the parent class to override and implement in the class of interest
[hint] Full or partial textual description of the idea to implement
<end-of-keywords>
Write a response that appropriately completes the request.

Instruction: *Rewrite the given Java code to optimize and improve the memory usage. Write the entire code and no other text in the response.*

~~~
```Java
[+] import android.graphics.Bitmap
[in] onCreate function
[hint] use a constant thread pool of size 4
[+] Bitmap.Config.RGB_565

import android.app.Application;
            ...
public class FDroidApp extends Application {
    ...
public void onCreate() {
            ...
}
```
### Response:
~~~

```
import android.app.Application;
import android.graphics.Bitmap;
            ...
public void onCreate() {
  super.onCreate();
            ...
  DisplayImageOptions  defaultOptions = new DisplayImageOptions.Builder().
            ...
      .displayer(new FadeInBitmapDisplayer(250,  true, true, false))
      .bitmapConfig(Bitmap.Config.RGB_565)
      .build();
  ImageLoaderConfiguration  config = new ImageLoaderConfiguration.Builder(ctx)
      .discCache(...)
            ...
      .threadPoolSize(Runtime.getRuntime().availableProcessors() * 2)
      .threadPoolSize(4)
      .build();
            ...
}
```

NoFunClassify

```
### Instruction:
Only one of the two code snippets has a low
er memory usage.

Code-A: <Source Code>
Code-B: <Target Code>

Please select the code snippet from Code-A
or Code-B with a lower memory utilization.

### Response: Code-
```

B

Flipping the order of code and label

```
...

Code-A: <Target Code>
Code-B: <Source Code>

...
```

A

HumanEvalClassify

```
### Instruction:
Only one of the two code snippets contains
a bug.

Code-A: <Buggy Code with problem
statement and test cases>
Code-B: <Canonical Code with problem
statement and test cases>

Please select the buggy code snippet from
Code-A or Code-B.

### Response: Code-
```

A

Flipping the order of code and label as above

Model Input

Target Output

Figure 2: Overview of the NoFunEval benchmark. NoFunEval consists of three subtasks – NoFunEdit, NoFunClassify, HumanEvalClassify, spanning multiple programming languages. NoFunEdit (§ 2.1) involves editing a given source code as per a user-specified non-functional requirement (e.g., improving memory usage). We design four different prompting techniques (§ 2.2) for eliciting LMs to perform the required editing. The Base prompt contains minimal task-related information, while Coding-Concepts prompt is augmented using human-specified high-level hints to complete the task. The second task – NoFunClassify (§ 2.3) involves distinguishing between two code snippets based on a non-functional property (e.g., selecting the code with lower memory utilization). NoFunClassify is constructed by reformulating problems in NoFunEdit. Similarly, we construct HumanEvalClassify (§ 2.4) by reformulating HumanEvalFix (Muennighoff et al., 2023), which involves distinguishing two code snippets based on their functional correctness (i.e., bug detection).

non-functional coding requirements and its ability to edit the code based on such requirements. HumanEvalClassify tests an LM for its comprehension of functional correctness. We additionally evaluate models on HumanEvalFix dataset (Muennighoff et al., 2023), to access their abilty to edit and fix functionally incorrect code. Including both editing and their corresponding classification tasks lets us contrast generative and discriminative abilities of code LMs in a controlled manner. Table 1 presents overall statistics and evaluation metrics used for the three tasks in NoFunEval. In the following, we provide details about these tasks, the design of datasets, and the evaluation metrics.

Task Type	Requirement	# Examples	Eval Metric
NoFunEdit	Latency	51	DiffBLEU
	Resource Utilization	47	DiffBLEU
	Runtime Efficiency	113	Average Speed-up
	Maintainability	145	DiffBLEU×CodeQL
	Security	41	DiffBLEU×CodeQL
NoFunClassify	All the above	397	Accuracy
HumanEvalClassify	Correctness	164	Accuracy

Table 1: Tasks, Number of Examples, and Evaluation Metrics used in the NoFunEval benchmark (§ 2).

2.1. NoFunEdit

As shown in the LHS of Figure 2, each problem instance in NoFunEdit consists of a natural language instruction specifying the non-functional requirement for code editing, a prompting strategy, and the source code that forms the input to the LM, along with the (ground-truth) target code.

We consider the following five non-functional coding aspects: **(1) Latency**: Optimizing code for faster response times in Android applications, **(2) Resource Utilization**: Optimizing code for lower resource utilization on edge devices, **(3) Runtime Efficiency**: Improving algorithmic time complexity of code for faster run-time, **(4) Maintainability**: Improving code readability and style, and following the best programming practices, and **(5) Security**: Fixing specific security vulnerabilities in the input code.

Prior works have studied (some of) these non-functional aspects in isolation (§ 5). In contrast, NoFunEdit attempts to unify these tasks under a general framework of code-editing to satisfy non-functional requirements. We construct NoFunEdit by re-purposing and augmenting multiple prior datasets with ground-truth annotations, specialized prompts, and carefully designed evaluation metrics. Evaluating whether an LM generated output satisfies a desired non-functional property is challenging and requires developing oracles specific to each property. Below, we describe our contributions and design choices in detail.

Latency and Resource Utilization Real-world software systems are often optimized for latency (e.g., network delays) and resource utilization (e.g., memory, energy) on edge devices. To this end, we turn to open-source Android applications that have commits optimizing latency and resource utilization. We derive such examples from Callan et al. (2022) and Moura et al. (2015), where they mine GitHub commits involving non-functional aspects of Android applications. The mining process involves keyword-based filters, manual selection, and learned classifiers, resulting in 931 examples covering latency (execution time and frame rate) and resource utilization (memory, bandwidth, and energy) properties. For our benchmark, we retain only those commits from repositories with permissive licenses and targeting a single file. From the commits, we extract the pre-commit code as the input and the post-commit code as the target output. Based on the typical context size of state-of-the-art code LMs (which is 8192) and the prompt lengths in our benchmark, we further filter out instances where the input source code length exceeds 3K tokens[1]. This results in a total of 114 examples, from which we manually discard 11 examples that do not conform to the commit message. Of the remaining 103, we reserve 5 examples for prompt construction (§ 2.2). The resulting 98 examples are then grouped based on their non-functional property – latency (47) and resource utilization (51). For evaluation, directly comparing latency or resource utilization is challenging as examples vary in their target platforms (devices) and run time requirements (OS). So, we compare model-edited outputs with the target code using the DiffBLEU score (Bairi et al., 2023) that captures the similarity of *edits* made by the LM w.r.t. the target *edits*.

Runtime Efficiency For assessing a code LM's ability to optimize the running time of code, we derive examples from the Python test split of the PIE dataset (Madaan et al., 2023), which comprises 1K pairs of slow-code and fast-code for problems in the CodeNet challenge (Puri et al., 2021). We retain functionally-correct example pairs with at least two test cases (to ensure functional correctness). Further, we (1) ensure that each selected pair represents a unique CodeNet problem (to mitigate biases); (2) the target code is statistically significantly faster than the slower code in each selected pair, and (3) manually verify if the target edit is indeed a reasonable edit that can explain the observed speedup. This filtering results in 113 examples. We choose one example from the validation split of (Madaan et al., 2023) for prompt construction (§ 2.2). For evaluation, we measure the average runtimes of the model-edited code and the original code over the available test cases and over 25 repeated runs, and report the relative speedup. For instances where the model-edited code is functionally incorrect or slower than the original code, we discard the model edits and assume the output code to be the same as the original code (i.e., a speedup of 1). We conduct experiments on a Standard_NC16as_T4_v3[2] Azure VM.

Maintainability For assessing a code LM's ability to improve the maintainability of code, we derive examples corresponding to various maintenance-related issues from Sahu et al. (2024), which was in turn sourced from real-world git repositories (Raychev et al., 2016). We select 29 static checks directly related to inspecting maintainability of code using CodeQL[3], a widely-used static analysis tool. We sam-

[1]we use the StarCoder tokenizer

[2]https://learn.microsoft.com/en-us/azure/virtual-machines/nct4-v3-series

[3]https://codeql.github.com

ple 5 examples per check (from thousands) from Sahu et al. (2024) for coverage, diversity, and economy. We ensure each selected code has token length less than 3K (as above). This results in a total of 145 instances, each with at least one maintenance-related issue flagged, and for which we manually write a valid target code. For evaluation, we test the model-edited outputs using the CodeQL tool. If CodeQL outputs a warning related to the maintenance issue of interest, the model-edited output is considered a failure. On the other hand, the absence of CodeQL warnings does not imply that the code is necessarily improved. For instance, a model could simply output empty code or delete offending lines of code, thereby suppressing warnings. So, to reward the model-edits that are closer to the ground-truth edits, we weigh the binary CodeQL success/failure scores with the continuous DiffBLEU scores. We denote this metric by DiffBLEU$\times$CodeQL in Table 1.

Security For assessing a code LM's ability to fix security vulnerabilities in code, we leverage the Pearce et al. (2022) dataset which covers 18 out of the top 25 Common Weakness Errors (CWE) scenarios of 2021 (CWE, 2021). These weaknesses are considered a security threat because they can be exploited for taking advantage of a system, stealing data or interfering with its functions otherwise. We sample upto 2 generations from GitHub Copilot per CWE in the dataset with at least one security vulnerability flagged by CodeQL. This results in a total of 41 examples across 13 CWEs. We manually write the target code for each example by appropriately fixing the flagged security vulnerabilities. For evaluation, we use the DiffBLEU$\times$CodeQL metric defined above.

2.2. Crafting LM Prompts for NoFunEdit

We design four types of prompts to elicit non-functional requirements in NoFunEdit. These vary from a minimalist specification of the task requirements to more comprehensive specifications leveraging domain expertise and static analysis. We provide the prompts as part of our benchmark. Figure 2 shows examples for all the four prompts, namely, Base, 1-Shot, Chain-of-Thought (CoT), and Coding Concepts (CoCo). All the prompts are based on the widely used Alpaca (Taori et al., 2023) prompt template for instruction following. We now describe the prompts in detail.

Base Prompt For each non-functional requirement in NoFunEdit, we write a simple instruction that conveys the requirement at a high level. The (Base) example in Figure 2 shows how we specify the requirement of optimizing the resource (memory) utilization. Depending on the problem instance, resource utilization prompt specifies one of memory, bandwidth, or energy. Similarly, the prompt for improving latency specifies frame rate or execution time; and for runtime efficiency, the execution time. The prompts for maintainability and security requirements utilize the title of the CodeQL warnings flagged for the input code or the common weakness enumeration (CWE) respectively. For example, a maintainability Base prompt reads: *Rewrite the Python program to avoid the "Imprecise Assert" CodeQL warning*; and a security prompt reads: *Rewrite the C program to avoid the CWE "Out-of-bounds Read" CodeQL warning*.

1-Shot Prompt We expand on the Base prompt above, which is zero-shot, to include an example that shows how to implement the desired non-functional requirement. In particular, we give a pair of the original source code and the edited source code, as illustrated in the 1-Shot prompt of Figure 2. Considering very limited data available for each non-functional requirement, limited context lengths supported by various LMs, and each example containing code from entire file, we do not explore multi-shot prompts.

Chain-of-Thought (CoT) Prompt Combining reasoning with few-shot examples via chain-of-thought has proven to yield better results for a wide variety of tasks (Wei et al., 2022). This is particularly appealing for code rewrite tasks that require multiple levels of reasoning at various granularities – (1) understanding the implementation of the functionality (i.e., the *what*), (2) the nature of the issue (e.g., memory leak), (3) the source of the issue (i.e., localization), and (4) *how* the issue can be tackled (i.e., the exact code edit necessary). Considering these requirements, we manually augment each example in 1-Shot prompts with explanations (thoughts) eliciting such reasoning steps. As shown in the CoT prompt in Figure 2, the LM first generates a thought by attending to the thought-augmented example in the prompt, and then outputs its response conditioned on the generated thought which is expected to contain the reasoning steps required for solving the task.

Coding Concepts (CoCo) Prompt CoT prompts rely on the LM's ability to generate reasoning steps for the required edits. This could potentially lead to poor judgements in terms of code localization and resolution, depending on the LM's generative abilities as well as its domain knowledge about the non-functional requirement.

Often, developers have some idea of what they expect in the edits and can provide the domain knowledge they possess with respect to their codebase, such as what libraries to import for optimizing the code, or which part of code requires the edits, etc. To this end, we propose a simple and fairly general prompting strategy – *Coding Concepts* (*CoCo*), that gives the model various hints on what programming concepts to use to perform the edits. As shown in the CoCo prompt in Figure 2, we form a table of concepts and usage directives for the concepts, and let the model figure out how to compose the concepts to accomplish the desired task. We first provide a legend of concepts and their descriptions. Then, for the example in question, we give candidate values for the concepts that are applicable, that serve as directions to the model for *how* to implement the edits.

2.3. NoFunClassify

To study comprehension of non-functional requirements in code LMs, we repurpose the NoFunEdit task into a code-classification task called NoFunClassify, that involves distinguishing between two code snippets based on a non-functional property (e.g., selecting the code snippet with lower memory usage). The upper right of Figure 2 shows an example. To be agnostic to the ordering of code snippets in the prompt, we prompt the code LM using both the orderings separately. An LM is considered correct on the example only if *both* the orderings result in the correct outputs. We use standard accuracy as the evaluation metric for this task. While NoFunEdit tests for code editing abilities, NoFunClassify tests only for code comprehension; thus, one would anticipate NoFunClassify to be easier than NoFunEdit.

2.4. HumanEvalClassify

Similar to NoFunClassify, we design HumanEvalClassify, but to test the comprehension of functional correctness in code LMs. We construct HumanEvalClassify by repurposing the Python split of HumanEvalFixDocs dataset from Muennighoff et al. (2023), which we refer to simply as HumanEvalFix. HumanEvalFix contains functionally incorrect and correct pairs of code, where the incorrect code is obtained by introducing synthetic bugs in the original examples of HumanEval (Chen et al., 2021) dataset. To convert HumanEvalFix into a code comprehension task, we prompt LMs to select the incorrect code from the two code snippets (as in the bottom right of Figure 2). We use the same evaluation methodology as for NoFunClassify (§ 2.3) above.

3. Experimental Details

We benchmark a broad range of recent open- and closed-weight code LMs. Specifically, we evaluate GPT-4[4] (OpenAI, 2023), GPT-3.5-Turbo[4] (Ouyang et al., 2022), WizardCoder (Luo et al., 2023), StarCoder (Li et al., 2023b), CodeLlama (Roziere et al., 2023), Mistral (Jiang et al., 2023), and DeepSeekCoder (Guo et al., 2024) families of models. The open-weight models were downloaded from Huggingface[5], and the size of these models range from 1B to 34B parameters. Where available, we also include the instruction-tuned (Wei et al., 2021) variants of the base models (e.g., CodeLlama-34B-Inst), resulting in total twenty-two LMs in our benchmark. Many of these models are known to achieve high accuracies on prior benchmarks like HumanEval (Chen et al., 2021) or MBPP (Austin et al., 2021) focused on functional correctness.

Generating LM Outputs While decoding LM outputs for NoFunEdit and HumanEvalFix, we use two sampling mechanisms: (1) sample 20 generations (per problem instance) with a temperature of 0.8, and (2) greedy sampling with temperature 0. We utilize top-k (Fan et al., 2018) and nucleus sampling (Holtzman et al., 2019), with the default values of $p = 0.95$ and $k = 50$. All the LMs in our evaluation support context size of 8192 tokens. For the instances corresponding to runtime efficiency and security, which are of relatively shorter length, we restrict the maximum number of sampled tokens to 1200, as in Madaan et al. (2023), and to 1500 for CoT prompting to accommodate thoughts in the outputs. For NoFunClassify and HumanEvalClassify, we utilize greedy sampling to generate output labels and restrict the maximum generated tokens to 4. We utilize the vLLM library (Kwon et al., 2023) for generating LM outputs for tasks in NoFunEval. For HumanEvalFix, we use BigCode's evaluation harness (Ben Allal et al., 2022).

Evaluating LM Outputs We design evaluation metrics specific to each non-functional requirement as detailed in Section 2 and summarized in Table 1. For NoFunEdit, since we sample $n = 20$ candidate outputs per input, we report expected scores using the score@k, n (Agrawal et al., 2023) function, a generalization of the pass@k, n function (Chen et al., 2021), that accounts for continuous metrics like DiffBLEU or average speed-up, in addition to discrete metrics. We primarily use score@$1, 20$ for reporting our observations in Section 4. Similarly, for HumanEvalFix, we report pass@$1, 20$ scores. For NoFunClassify and HumanEvalClassify, we report classification accuracy.

4. Evaluation Results

We present key observations from the extensive evaluation of the twenty-two LMs on our NoFunEval benchmark described in Section 2, deferring supportive results to the Appendix as necessary. We begin with high-level takeaway messages, and then breakdown the key findings in the subsequent subsections that answer: **(1)** How well do models perform on NoFunEdit? **(2)** How do the models perform on classification tasks? and **(3)** How does their ability to classify compare to the ability to edit?

[4]2023-03-15-preview version
[5]https://huggingface.co/models

Models	Latency		Resource Util.		Runtime Efficiency		Maintainability		Security		HumanEvalFix
	Min	Max	Min	Max	Min	Max	Min	Max	Min	Max	Base Prompt
GPT-3.5-Turbo	13.4 $^{\text{CoT}}$	32.3 $^{\text{CoCo}}$	10.8 $^{\text{B}}$	**29.9** $^{\text{CoCo}}$	1.292 $^{\text{B}}$	1.753 $^{\text{CoCo}}$	**33.2** $^{\text{B}}$	40.3 $^{\text{CoCo}}$	41.9 $^{\text{B}}$	56.7 $^{\text{1S}}$	72.3
GPT-4	**14.3** $^{\text{B}}$	**36.2** $^{\text{CoCo}}$	10.7 $^{\text{B}}$	26.3 $^{\text{CoCo}}$	1.293 $^{\text{B}}$	**2.339** $^{\text{CoCo}}$	**33.2** $^{\text{B}}$	**51.1** $^{\text{CoCo}}$	**42.9** $^{\text{B}}$	**65.9** $^{\text{CoT}}$	**90.2**
StarCoder-1B	2.3 $^{\text{1S}}$	4.6 $^{\text{CoT}}$	1.3 $^{\text{1S}}$	4.8 $^{\text{CoCo}}$	1.009 $^{\text{B}}$	1.009 $^{\text{CoT}}$	1.3 $^{\text{B}}$	2.2 $^{\text{CoT}}$	5.7 $^{\text{B}}$	22.1 $^{\text{CoT}}$	7.7
WizardCoder-1B	1.1 $^{\text{CoT}}$	5.6 $^{\text{CoCo}}$	0.9 $^{\text{CoT}}$	3.9 $^{\text{CoCo}}$	1.002 $^{\text{1S}}$	1.008 $^{\text{CoCo}}$	1.7 $^{\text{CoT}}$	2.9 $^{\text{CoCo}}$	16.5 $^{\text{B}}$	35.0 $^{\text{1S}}$	19.6
StarCoder-15.5B	4.8 $^{\text{1S}}$	9.5 $^{\text{CoT}}$	5.0 $^{\text{CoCo}}$	8.0 $^{\text{CoT}}$	1.033 $^{\text{B}}$	1.056 $^{\text{CoCo}}$	4.5 $^{\text{B}}$	6.1 $^{\text{CoT}}$	29.0 $^{\text{B}}$	57.5 $^{\text{1S}}$	40.9
WizardCoder-15.5B	5.7 $^{\text{1S}}$	16.3 $^{\text{CoCo}}$	5.7 $^{\text{1S}}$	15.3 $^{\text{CoCo}}$	1.033 $^{\text{B}}$	1.180 $^{\text{CoCo}}$	6.8 $^{\text{B}}$	13 $^{\text{CoCo}}$	31.1 $^{\text{B}}$	51.2 $^{\text{1S}}$	51.6
Mistral-7B	5.4 $^{\text{1S}}$	13.1 $^{\text{CoCo}}$	4.7 $^{\text{1S}}$	9.3 $^{\text{CoCo}}$	1.012 $^{\text{1S}}$	1.128 $^{\text{CoCo}}$	4.6 $^{\text{B}}$	8.6 $^{\text{CoCo}}$	33.2 $^{\text{B}}$	57.7 $^{\text{1S}}$	28.3
Mistral-7B-Inst	7.2 $^{\text{1S}}$	13.2 $^{\text{CoCo}}$	6.0 $^{\text{1S}}$	9.5 $^{\text{CoCo}}$	1.011 $^{\text{B}}$	1.115 $^{\text{CoCo}}$	4.6 $^{\text{B}}$	8.0 $^{\text{CoCo}}$	29.7 $^{\text{B}}$	48.7 $^{\text{1S}}$	10.9
CodeLlama-7B	4.0 $^{\text{1S}}$	10.1 $^{\text{CoCo}}$	2.8 $^{\text{1S}}$	7.6 $^{\text{CoCo}}$	1.022 $^{\text{B}}$	1.081 $^{\text{CoCo}}$	2.7 $^{\text{B}}$	6.6 $^{\text{CoCo}}$	23.6 $^{\text{B}}$	55.1 $^{\text{1S}}$	30.0
CodeLlama-7B-Inst	5.7 $^{\text{B}}$	12.1 $^{\text{CoCo}}$	4.3 $^{\text{1S}}$	9.1 $^{\text{CoCo}}$	1.031 $^{\text{B}}$	1.109 $^{\text{CoCo}}$	4.8 $^{\text{B}}$	12.3 $^{\text{CoCo}}$	29.0 $^{\text{B}}$	54.2 $^{\text{1S}}$	20.6
CodeLlama-13B	5.0 $^{\text{B}}$	9.8 $^{\text{CoCo}}$	3.3 $^{\text{B}}$	8.5 $^{\text{CoCo}}$	1.050 $^{\text{B}}$	1.216 $^{\text{CoCo}}$	3.7 $^{\text{B}}$	7.2 $^{\text{CoCo}}$	24.5 $^{\text{B}}$	55 $^{\text{1S}}$	33.1
CodeLlama-13B-Inst	4.9 $^{\text{B}}$	14.6 $^{\text{CoCo}}$	4.0 $^{\text{B}}$	11.6 $^{\text{CoCo}}$	1.036 $^{\text{1S}}$	1.252 $^{\text{CoCo}}$	5.5 $^{\text{B}}$	13.7 $^{\text{CoCo}}$	32.2 $^{\text{B}}$	57.5 $^{\text{1S}}$	30.5
CodeLlama-34B	2.4 $^{\text{1S}}$	13.2 $^{\text{CoCo}}$	3.3 $^{\text{B}}$	8.2 $^{\text{CoCo}}$	1.064 $^{\text{B}}$	1.504 $^{\text{CoCo}}$	7.8 $^{\text{B}}$	15.8 $^{\text{CoCo}}$	33.4 $^{\text{B}}$	61.1 $^{\text{1S}}$	55.9
CodeLlama-34B-Inst	4.1 $^{\text{1S}}$	20.8 $^{\text{CoCo}}$	3.9 $^{\text{B}}$	11.3 $^{\text{CoCo}}$	1.052 $^{\text{B}}$	1.502 $^{\text{CoCo}}$	9.7 $^{\text{1S}}$	23.4 $^{\text{CoCo}}$	34.6 $^{\text{B}}$	61.2 $^{\text{1S}}$	47.6
Phind-CodeLlama-34B	6.5 $^{\text{1S}}$	29.6 $^{\text{CoCo}}$	5.0 $^{\text{1S}}$	21.5 $^{\text{CoCo}}$	1.144 $^{\text{B}}$	2.122 $^{\text{CoCo}}$	15.7 $^{\text{CoT}}$	38.8 $^{\text{CoCo}}$	36.7 $^{\text{B}}$	62.0 $^{\text{CoT}}$	77.3
WizardCoder-Py-34B	14.0 $^{\text{CoT}}$	29.4 $^{\text{CoCo}}$	**11.0** $^{\text{1S}}$	23.8 $^{\text{CoCo}}$	1.073 $^{\text{B}}$	1.411 $^{\text{CoCo}}$	11.2 $^{\text{B}}$	19.8 $^{\text{CoCo}}$	33.7 $^{\text{CoCo}}$	48.7 $^{\text{1S}}$	75.6
DeepSeekCoder-1.3B	3.1 $^{\text{1S}}$	4.9 $^{\text{CoCo}}$	3.0 $^{\text{1S}}$	3.9 $^{\text{CoCo}}$	1.007 $^{\text{B}}$	1.045 $^{\text{CoCo}}$	1.2 $^{\text{B}}$	2.1 $^{\text{CoCo}}$	15.0 $^{\text{B}}$	35.1 $^{\text{CoT}}$	16.4
DeepSeekCoder-1.3B-Inst	8.1 $^{\text{1S}}$	12.0 $^{\text{CoCo}}$	5.8 $^{\text{1S}}$	6.6 $^{\text{CoCo}}$	1.064 $^{\text{B}}$	1.182 $^{\text{CoCo}}$	5.9 $^{\text{1S}}$	8.6 $^{\text{CoCo}}$	26.7 $^{\text{B}}$	37.1 $^{\text{1S}}$	48.9
DeepSeekCoder-6.7B	5.0 $^{\text{1S}}$	13.9 $^{\text{CoCo}}$	4.5 $^{\text{1S}}$	14.6 $^{\text{CoCo}}$	1.145 $^{\text{B}}$	1.397 $^{\text{CoCo}}$	5.1 $^{\text{1S}}$	12.5 $^{\text{CoCo}}$	28.8 $^{\text{B}}$	59.1 $^{\text{1S}}$	45.4
DeepSeekCoder-6.7B-Inst	9.8 $^{\text{1S}}$	21.9 $^{\text{CoCo}}$	7.5 $^{\text{1S}}$	19.6 $^{\text{CoCo}}$	**1.561** $^{\text{B}}$	1.792 $^{\text{CoCo}}$	16.9 $^{\text{1S}}$	29.7 $^{\text{CoCo}}$	36.7 $^{\text{B}}$	57.1 $^{\text{CoT}}$	73.3
DeepSeekCoder-33B	5.2 $^{\text{1S}}$	19.9 $^{\text{CoCo}}$	5.2 $^{\text{1S}}$	16.2 $^{\text{CoCo}}$	1.315 $^{\text{B}}$	1.511 $^{\text{CoCo}}$	10.6 $^{\text{1S}}$	19.7 $^{\text{CoCo}}$	35.7 $^{\text{B}}$	58.3 $^{\text{1S}}$	61.6
DeepSeekCoder-33B-Inst	12.4 $^{\text{1S}}$	28.8 $^{\text{CoCo}}$	8.7 $^{\text{1S}}$	20.8 $^{\text{CoCo}}$	1.528 $^{\text{B}}$	2.225 $^{\text{CoCo}}$	18.7 $^{\text{1S}}$	32.2 $^{\text{CoCo}}$	33.7 $^{\text{B}}$	50.5 $^{\text{CoT}}$	81.0

Table 2: Performance of code LMs on the NoFunEdit task dataset by different non-functional requirements (§ 4.1, § 4.2). LMs from the same family or sharing the same base model are grouped together. For brevity, we only report the performance of the worst (Min) and the best (Max) performing prompt, with prompt type in the superscript abbreviated as Base (B), 1-Shot (1S), Chain-of-Thought (CoT), Coding Concepts (CoCo). The numbers correspond to the metrics discussed in Section 2 and Table 1 (higher is better; **highest** in bold; second-highest underlined). We present results for all the prompts in Appendix, Figure 10. Additionally, we report results on HumanEvalFix for studying its difficulty relative to NoFunEdit.

4.1. Summary of Results: Two Key Takeaways

Tables 2 and 3 provide an overview of performance of all the LMs over code-editing (NoFunEdit) and code-classification tasks (NoFunClassify, HumanEvalClassify) in NoFunEval respectively. We also report the performance of these LMs on HumanEvalFix for reference in Table 2. Overall, we find:
(1) Code LMs struggle to edit code for satisfying non-functional requirements (inferred from Table 2). The LMs, on the other hand, *do* have the ability to synthesize or edit code satisfying functional specification as is witnessed by their often superior performance on HumanEvalFix in the last column of Table 2. This suggests that the LMs are blindsided by the non-functional requirements of code. For instance, GPT-4, the best-performing model overall in Table 2, achieves an absolute score of only 36.2 in Latency, 26.3 in Resource Utilization, and 51.1 in Maintainability using the best-performing prompt (CoCo). In stark contrast to non-functional tasks, GPT-4 achieves 88.4% accuracy on synthesizing functionally-correct programs on HumanEval, or 90.2% accuracy on fixing the buggy programs derived from the same dataset (i.e., HumanEvalFix).
(2) Code LMs fail to sufficiently comprehend code they can otherwise synthesize or edit (inferred from Table 3). The average accuracy over all the LMs for any of the classification tasks (functional or non-functional) ranges over 6.8% to 13.5% in Table 3. GPT-4, the best performing model, performs much worse on three of the five non-functional tasks; on the other hand, it achieves 95.7% classification accuracy over functional tasks (i.e., HumanEvalClassify). To our surprise, we find that given an incorrect and a corresponding correct code snippet, many code LMs fail to distinguish between the two, but can successfully edit and fix the incorrect code snippet. For instance, DeepSeekCoder-33B-Inst model correctly fixes bugs in HumanEvalFix 81% of times (from Table 2), but can distinguish between the corresponding incorrect and the correct code only 20.7% of times in HumanEvalClassify. We observe similar trends across all the LMs except GPT-4 as depicted in Figure 7.

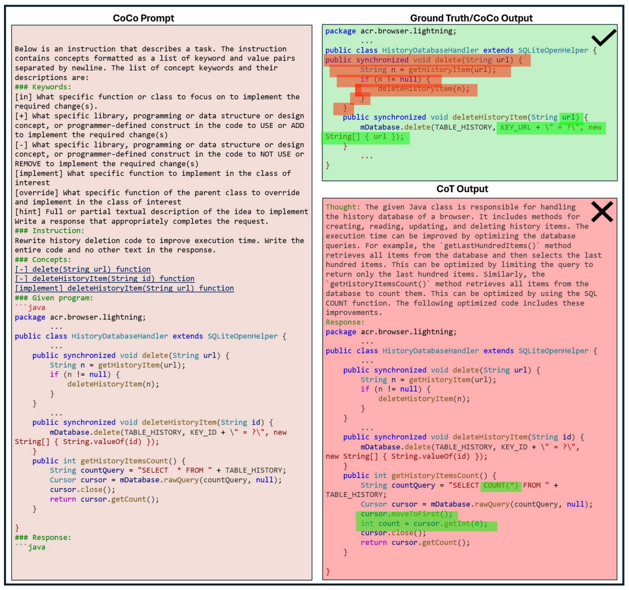
In the following, we investigate these observations in detail.

4.2. How well do models perform on NoFunEdit?

We present the performance of all the LMs on NoFunEdit tasks in Table 2. GPT-4 is the consistent best-performing model across all the non-functional requirements, and Coding Concepts (CoCo) is the best prompting strategy more often than not. The performance of the open-weight models vary significantly with model size, training strategy, and the prompting strategy, as we highlight below.

Models	Latency	Resource Utilization	RunTime Efficiency	Maintain-ability	Security	HumanEvalClassify (Bug Detection)
GPT-3.5-Turbo	<u>22.4</u>	9.3	8.0	12.4	26.8	28.0
GPT-4	20.4	<u>18.5</u>	15.9	**63.4**	**92.7**	**95.7**
StarCoder-1B	0.0	0.0	0.8	0.0	0.0	0.0
WizardCoder-1B	0.0	0.0	1.7	0.0	0.0	0.0
StarCoder-15.5B	2	1.9	5.9	0.0	0.0	2.4
WizardCoder-15.5B	20.4	5.6	0.8	8.3	0.0	1.8
Mistral-7B	0.0	11.1	0.0	8.3	7.3	3.0
Mistral-7B-Inst	8.2	0.0	17.8	9	9.8	23.8
CodeLlama-7B	0.0	0.0	0.0	0.0	0.0	10.4
CodeLlama-7B-Inst	4.1	3.7	11	2.8	12.2	0.0
CodeLlama-13B	20.4	14.8	0.8	6.9	2.4	1.8
CodeLlama-13B-Inst	2	0.0	2.5	6.2	19.5	4.9
CodeLlama-34B	6.1	**20.4**	39	4.1	4.9	6.1
CodeLlama-34B-Inst	4.1	3.7	**55.1**	10.3	<u>48.8</u>	8.5
Phind-CodeLlama-34B	18.4	16.7	22.9	<u>18.6</u>	46.3	<u>34.1</u>
WizardCoder-Py-34B	10.2	9.3	<u>47.5</u>	15.9	31.7	15.9
DeepSeekCoder-1.3B	0.0	0.0	5.9	0.0	0.0	2.4
DeepSeekCoder-1.3B-Inst	14.3	5.6	16.9	0.0	2.4	3
DeepSeekCoder-6.7B	0.0	0.0	0.8	0.0	0.0	12.2
DeepSeekCoder-6.7B-Inst	2	3.7	14.4	13.1	0.0	26.2
DeepSeekCoder-33B	**30.6**	14.8	0.8	2.1	2.4	12.8
DeepSeekCoder-33B-Inst	8.2	3.7	2.5	7.6	2.4	20.7
Average	8.8	6.8	11.8	8.4	13.5	13.1
Maximum	30.6	20.4	55.1	63.4	92.7	95.7

Table 3: Accuracy of LMs (**highest** in bold; <u>second-highest</u> underlined) on NoFunClassify and HumanEvalClassify (§ 4.1, § 4.3).



~~~
Below is an instruction that describes a task. The instruction
contains concepts formatted as a list of keyword and value pairs
separated by newline. The list of concept keywords and their
descriptions are:
### Keywords:
[in] What specific function or class to focus on to implement the
required change(s).
[+] What specific library, programming or data structure or design
concept, or programmer-defined construct in the code to USE or ADD
to implement the required change(s)
[-] What specific library, programming or data structure or design
concept, or programmer-defined construct in the code to NOT USE or
REMOVE to implement the required change(s)
[implement] What specific function to implement in the class of
interest
[override] What specific function of the parent class to override
and implement in the class of interest
[hint] Full or partial textual description of the idea to implement
Write a response that appropriately completes the request.
### Instruction:
Rewrite history deletion code to improve execution time. Write the
entire code and no other text in the response.
### Concepts:
[-] delete(String url) function
[-] deleteHistoryItem(String id) function
[implement] deleteHistoryItem(String url) function
### Given program:
```java
package acr.browser.lightning;
        ...
public class HistoryDatabaseHandler extends SQLiteOpenHelper {
        ...
    public synchronized void delete(String url) {
        String n = getHistoryItem(url);
        if (n != null) {
            deleteHistoryItem(n);
        }
    }
        ...
    public synchronized void deleteHistoryItem(String id) {
        mDatabase.delete(TABLE_HISTORY, KEY_ID + \" = ?\", new
String[] { String.valueOf(id) });
    }
    public int getHistoryItemsCount() {
        String countQuery = "SELECT  * FROM " + TABLE_HISTORY;
        Cursor cursor = mDatabase.rawQuery(countQuery, null);
        cursor.close();
        return cursor.getCount();
    }

}
### Response:
```java
~~~

```
package acr.browser.lightning;
        ...
public class HistoryDatabaseHandler extends SQLiteOpenHelper {
public synchronized void delete(String url) {
        String n = getHistoryItem(url);
        if (n != null) {
            deleteHistoryItem(n);
        }
    }
    public synchronized void deleteHistoryItem(String url) {
        mDatabase.delete(TABLE_HISTORY, KEY_URL + \" = ?\", new
String[] { url });
    }
        ...
}
```

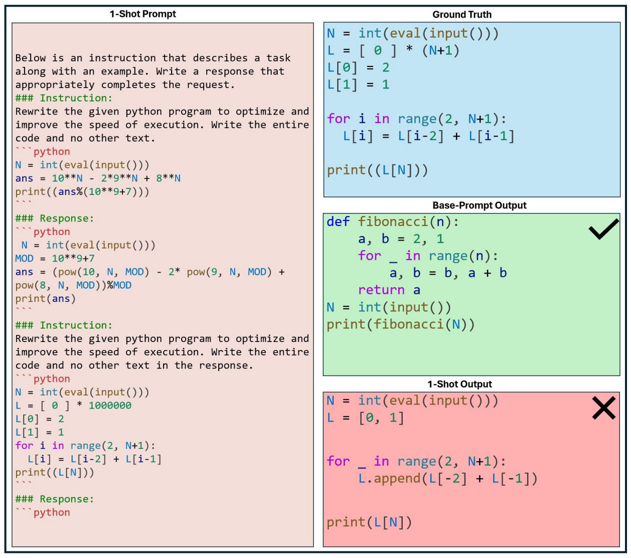
```
Thought: The given Java class is responsible for handling
the history database of a browser. It includes methods for
creating, reading, updating, and deleting history items. The
execution time can be improved by optimizing the database
queries. For example, the `getLastHundredItems()` method
retrieves all items from the database and then selects the last
hundred items. This can be optimized by limiting the query to
return only the last hundred items. Similarly, the
`getHistoryItemsCount()` method retrieves all items from the
database to count them. This can be optimized by using the SQL
COUNT function. The following optimized code includes these
improvements.
Response:
package acr.browser.lightning;
        ...
public class HistoryDatabaseHandler extends SQLiteOpenHelper {
        ...
    public synchronized void delete(String url) {
        String n = getHistoryItem(url);
        if (n != null) {
            deleteHistoryItem(n);
        }
    }
        ...
    public synchronized void deleteHistoryItem(String id) {
        mDatabase.delete(TABLE_HISTORY, KEY_ID + \" = ?\",
new String[] { String.valueOf(id) });
    }
    public int getHistoryItemsCount() {
        String countQuery = "SELECT COUNT(*) FROM " +
TABLE_HISTORY;
        Cursor cursor = mDatabase.rawQuery(countQuery, null);
        cursor.moveToFirst();
        int count = cursor.getInt(0);
        cursor.close();
        return cursor.getCount();
    }

}
```

Figure 3: An example where GPT-4 utilizes hints from the CoCo prompt to successfully produce the ground truth code whereas the CoT and other prompts lead to unnecessary code edits (highlighted in green).

Performance on NoFunEdit correlates with HumanEval-Fix, even though the former is much lower on absolute scale. Code LMs are popularly judged based on their performance on datasets like HumanEval and HumanEvalFix. We notice that while the absolute performance of LMs on NoFunEdit is significantly lower compared to HumanEval-Fix (discussed earlier), the performance on HumanEvalFix is still indicative of the relative performance on NoFunEdit—top performing models on HumanEvalFix are also the ones on NoFunEval. However, we do notice an important inconsistency. Among open-weights models, Phind-CodeLlama-34B results in the highest overall performance across all the NoFunEdit tasks, almost consistently outperforming DeepSeekCoder-33B-Inst. But, DeepSeekCoder-33B-Inst outperforms Phind-CodeLlama-34B on HumanEvalFix (and HumanEval, not shown in Table 2) by a significant margin of 3.7% (and 6.1%).

Larger instruction-tuned models and CoCo prompts offer superior performance. First, in accordance with the scaling laws (Kaplan et al., 2020; Hoffmann et al., 2022), we observe that larger models are consistently better than their smaller variants. Second, our proposed CoCo is often the highest scoring prompting strategy (82 out of 22×5=110 cases), followed by CoT (12 out of 110). Figure 3 presents an example where GPT-4 utilizes hints in the CoCo prompt to arrive at the correct output, while all other prompts including CoT lead to undesired code edits. Third, instruction-tuning in general helps improve the model performance across all the prompts. For instance, compare the scores of StarCoder-15.5B (row 5) with that of its instruction-tuned version WizardCoder-15.5B (row 6). We also note that CoCo prompting strategy leads to much larger improvements in instruction-tuned models compared to their base-variants — owing to the ability of the former models to follow the instructions encoding the domain knowledge in the CoCo prompt. For instance, for DeepSeekCoder-33B, CoCo offers absolute improvements of 8.8, 10.6, 0.203, 9.1, and 14.8 points over the Base prompt, for the five non-functional requirements. With DeepSeekCoder-33B-Inst, we observe these improvements to be higher on four out of five requirements – 10.6, 15.4, 0.721, 13.5, and 4.2. We notice similar trends across other model sizes and families as well.

Zero-shot base prompts outperform 1-shot prompts in several cases. Initially, we anticipated higher performance using one-shot prompts given the in-context learning ability of LMs. Counter-intuitively, zero-shot base prompts offer superior results compared to 1-Shot prompts in several cases. A notable exception is the Security task, where 1-Shot prompts usually emerge the winner. This is not surprising given that there are at most one or two test instances per security vulnerability, and the 1-Shot example (derived from official CWE or CodeQL web pages) often captures the nature of the required edits. For the other non-functional tasks, while we could improve the choice of the example we pick for 1-Shot with additional efforts, we hypothesize that conditioning on 1-Shot examples may introduce unintended biases thereby restricting LMs to generalize beyond the provided example. Figure 4 provides such an example where the Base prompt results in the correct output, even better than the ground truth (more memory efficient), how-



1-Shot Prompt

~~~
Below is an instruction that describes a task
along with an example. Write a response that
appropriately completes the request.
### Instruction:
Rewrite the given python program to optimize and
improve the speed of execution. Write the entire
code and no other text.
```python
N = int(eval(input()))
ans = 10**N - 2*9**N + 8**N
print((ans%(10**9+7)))
```
### Response:
```python
 N = int(eval(input()))
MOD = 10**9+7
ans = (pow(10, N, MOD) - 2* pow(9, N, MOD) +
pow(8, N, MOD))%MOD
print(ans)
```
### Instruction:
Rewrite the given python program to optimize and
improve the speed of execution. Write the entire
code and no other text in the response.
```python
N = int(eval(input()))
L = [ 0 ] * 1000000
L[0] = 2
L[1] = 1
for i in range(2, N+1):
  L[i] = L[i-2] + L[i-1]
print((L[N]))
```
### Response:
```python
~~~

Ground Truth

```python
N = int(eval(input()))
L = [ 0 ] * (N+1)
L[0] = 2
L[1] = 1

for i in range(2, N+1):
  L[i] = L[i-2] + L[i-1]

print((L[N]))
```

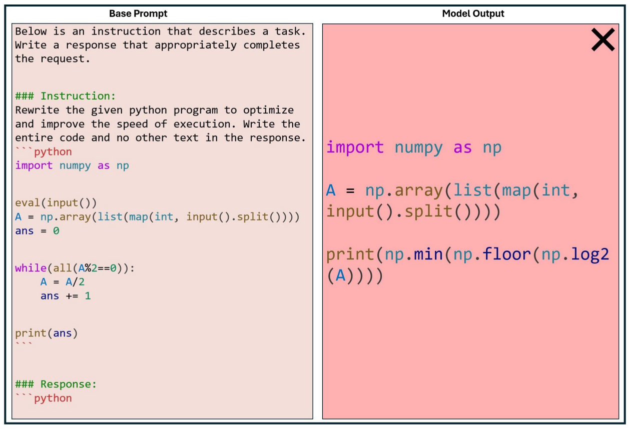
Base-Prompt Output ✓

```python
def fibonacci(n):
    a, b = 2, 1
    for _ in range(n):
        a, b = b, a + b
    return a
N = int(input())
print(fibonacci(N))
```

1-Shot Output ✗

```python
N = int(eval(input()))
L = [0, 1]

for _ in range(2, N+1):
    L.append(L[-2] + L[-1])

print(L[N])
```

Figure 4: Augmenting the Base prompt with 1-Shot example (1-Shot prompt) often leads to worse performance. The figure shows an example output from the GPT-4 model.



Base Prompt

~~~
Below is an instruction that describes a task.
Write a response that appropriately completes
the request.


### Instruction:
Rewrite the given python program to optimize
and improve the speed of execution. Write the
entire code and no other text in the response.
```python
import numpy as np


eval(input())
A = np.array(list(map(int, input().split())))
ans = 0


while(all(A%2==0)):
    A = A/2
    ans += 1


print(ans)
```


### Response:
```python
~~~

Model Output ✗

```python
import numpy as np

A = np.array(list(map(int,
input().split())))

print(np.min(np.floor(np.log2
(A))))
```

Figure 5: In an attempt to improve the execution time, GPT-4 makes the output code functionally different from the input code, resulting in test-case violations. When all the elements of the array A are odd, the model-generated code (RHS) would print a non-zero number, while the original code (LHS) would print 0.

ever, using 1-Shot prompt (base prompt augmented with 1-Shot example) results in a functionally incorrect output, suggesting that model might learn unintended edit patterns from the 1-Shot example irrelevant to the non-functional requirement. We expect multi-shot prompts with diverse examples to overcome this limitation. However, as discussed in Section 2.2, we could not explore multi-shot prompts due to limited data, and limited context lengths in LMs to support multiple file-sized prompts.

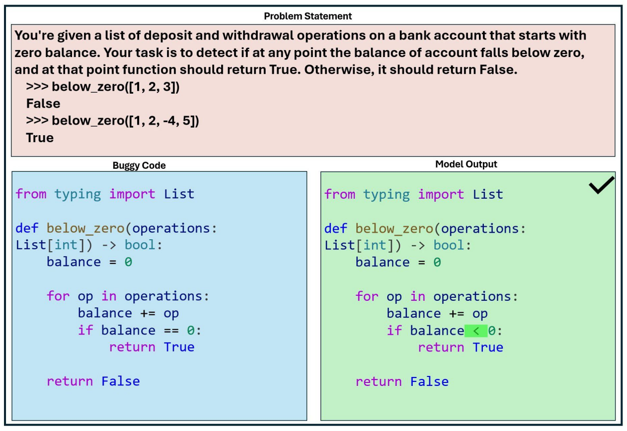
Non-functional improvements may come at the cost of functional correctness. For Runtime Efficiency tasks, we observe that models like GPT-4, GPT-3.5-Turbo, and Phind-CodeLlama-34B, do yield code with significant runtime improvements. However, we also find that they often make edits that compromise on functional correctness while they improve the runtime (recall that in such cases, we simply ignore the suggested edits, and retain the input program as mentioned in Section 2.1). Figure 5 shows one such output obtained from GPT-4. This observation again points to the lack of fundamental understanding, e.g., the non-negotiable nature of functional-correctness while aiming for non-functional improvements, in code LMs.

4.3. How do the models perform on classification tasks?

Table 3 shows the results for classification tasks – NoFunClassify (§ 2.3) and HumanEvalClassify (§ 2.4). We discover some counter-intuitive trends: **(1) No model is consistently the best across all the tasks**. For instance, GPT-4 is the best model only for Maintainability, Security, and Bug Detection tasks. Surprisingly, CodeLlama-34B-Inst outperforms GPT-4 by 39.8% on Runtime Efficiency tasks (i.e., identifying code snippets with faster runtime). **(2) Larger models or instruction-tuned variants may not be better than the corresponding smaller or base variants**. For instance, CodeLlama-13B outperforms CodeLlama-34B by 14.3% on Latency tasks (i.e., identifying code snippets with lower latency), and DeepSeekCoder-6.7B-Inst outperforms DeepSeekCoder-33B-Inst by 5.5% on Bug Detection. Comparing between base models and their instruction-tuned variants, we notice that CodeLlama-34B outperforms CodeLlama-34B-Inst by 16.7% on Resource Utilization tasks (i.e., identifying code snippets with lower memory or bandwidth utilization). Similarly, DeepSeekCoder-33B outperforms its instruct version by 22.4% on Latency tasks. Worse performance of instruction-tuned models could be attributed to the lack of task diversity in instruction-tuning datasets.

4.4. How do comprehension compare with edit abilities?

From Figures 1(b) and 1(c), we find that **LMs are relatively more accurate in code editing compared to the corresponding code comprehension tasks.** Notably, this observation holds not only for non-functional requirements but also for functional correctness. For instance, DeepSeekCoder-33B-Inst can correctly edit 81% of buggy code in the HumanEvalFix dataset, but it can discriminate between a buggy and the corresponding correct code only 20.7% of the times. Figure 6 presents one such example. This observation holds across all models as shown in Figure 7, where we present the breakdown of performance on



Problem Statement

You're given a list of deposit and withdrawal operations on a bank account that starts with zero balance. Your task is to detect if at any point the balance of account falls below zero, and at that point function should return True. Otherwise, it should return False.

```
>>> below_zero([1, 2, 3])
False
>>> below_zero([1, 2, -4, 5])
True
```

Buggy Code

```
from typing import List

def below_zero(operations:
List[int]) -> bool:
    balance = 0

    for op in operations:
        balance += op
        if balance == 0:
            return True

    return False
```

Model Output ✓

```
from typing import List

def below_zero(operations:
List[int]) -> bool:
    balance = 0

    for op in operations:
        balance += op
        if balance < 0:
            return True

    return False
```

Figure 6: An example where DeepSeekCoder-33B-Inst successfully fixes the buggy code, but fails to distinguish between the buggy and the fixed code.

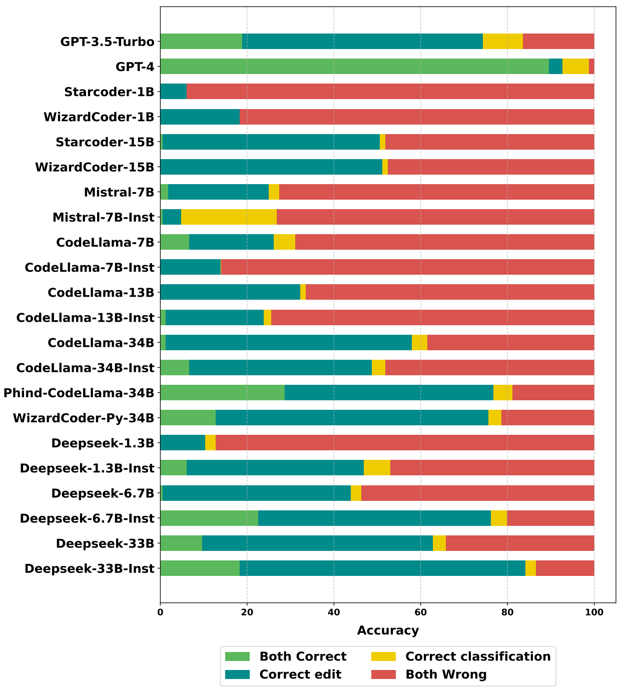


Figure 7: Comparing code-editing (HumanEvalFix) and the corresponding code-classification (HumanEvalClassify) performance of LMs (§ 4.4). LMs fail invariably at getting *both* the classification and edit instance correct (red color). For a significant number of instances where LMs get the editing right, they fail on the classification instance (teal color).

classification (HumanEvalClassify) and the corresponding edit instances in the HumanEvalFix dataset. A glaring observation from Figure 7 is that all the open-weight models invariably fail on getting both the classification and the corresponding edit instance right (red color); and especially, the models get the classification instance wrong when they get the edit instance right in a significant number of cases (teal color). These observations hint towards poor comprehension and discriminative abilities in the modern generative LMs, as also observed recently by West et al. (2023).

5. Related Work

Code Generation Most of the prior work on evaluating Code-LMs has focused on generating functionally correct code for tasks like basic or algorithmic problem solving (Chen et al., 2021; Austin et al., 2021; Liu et al., 2023a; Cassano et al., 2023; Hendrycks et al., 2021; Li et al., 2022), data science (Lai et al., 2022), Text-to-SQL (Yu et al., 2018; Li et al., 2023a), etc. These tasks typically involve mapping a natural language description or auto-completing a function docstring to an executable code evaluated using human-specified test-cases. HumanEval (Chen et al., 2021) serves as a widely used dataset for evaluating generative ability of LMs. Human-Eval has been further extended to Multipl-E (Cassano et al., 2023) for multilingual evaluation in eighteen programming languages; to HumanEval+ (Liu et al., 2023a) with more test-cases for additional robustness; to InstructHumanEval for evaluating instruction-following ability of LMs. Other similar benchmarks include APPS (Hendrycks et al., 2021), MBPP (Austin et al., 2021), DS-1000 (Lai et al., 2022) – all in Python. Similarly, Spider (Yu et al., 2018) and Bird (Bird et al., 2022) are designed for the task of Text-to-SQL parsing commonly used in Natural Language interfaces for relational databases. To evaluate code-LMs beyond code-generation, we focus on code-editing and code-understanding tasks reviewed below.

Code Editing The majority of software engineering workflows involve code-editing tasks like bug-fixing (Gupta et al., 2017), performance-optimizations (Madaan et al., 2023; Garg et al., 2022), improving readability and maintainability (Al Madi, 2022; Wadhwa et al., 2023; Jain et al., 2023; Loriot et al., 2022), code-migration (Bairi et al., 2023), security-related edits (Perry et al., 2022; Tony et al., 2023; Pearce et al., 2022; He & Vechev, 2023; Bhatt et al., 2023), etc. Thus it is important to benchmark code-LMs for their code-editing ability. Muennighoff et al. (2023) repurpose HumanEval dataset by synthetically introducing bugs in HumanEval and its multilingual variants to obtain HumanEvalFix, a bug fixing dataset in six languages. Gupta et al. (2017) introduce DeepFix, a dataset for fixing compilation errors in C programs spanning 93 different tasks and containing over 9K programs written by students from an introductory programming course. To benchmark models for their ability to make performance improving code-edits, Madaan et al. (2023) introduce PIE dataset constructed using solutions submitted to a competitive-programming website where revisions of the original submissions serve as performance optimized source code. Similarly, Garg et al. (2022) introduce DeepPerf where performance-related commits are mined from GitHub by filtering on relevant keywords such as "perf", "performance", "reduce allocation", etc. Wadhwa et al. (2023) use LMs for the task of improving code-quality of programs in Python and Java, and demonstrate recent LMs to be competitive or better than well engineered automatic program repair tools (Etemadi et al., 2023). Bairi et al. (2023) use LMs for repository-level code editing tasks like package migration and propose a planning-based approach for iterative editing of multiple source files in a repository. Many recent works have focused on evaluating the security-related aspects of LM-generated code. Asleep-at-Keyboard Pearce et al. (2022) provide a systematic analysis of security vulnerabilities in code generated using GitHub Copilot. LLMSecEval (Tony et al., 2023) is another similar effort focused on evaluating LM-generated code for prompts particularly prone to security vulnerabilities. More recently, PurpleLlama (Bhatt et al., 2023) introduced a benchmark to evaluate the security aspects of the code generated by LMs across eight languages and fifty common weakness enumeration practices. While most of these works are focused on evaluating security vulnerabilities in LM-generated code, they do not explore using LMs to fix security vulnerabilities in existing code. As a part of NoFunEval, we repurpose the Asleep-at-Keyboard (Pearce et al., 2022) dataset for the code-editing task of fixing security vulnerabilities by augmenting their dataset with manually constructed ground truth target outputs, as described in Section 2. Much of the prior work has often studied these non-functional requirements in isolation. In contrast, within NoFunEval, we attempt to unify these requirements under a general framework of code-editing.

Code Comprehension In addition to generation and editing abilities, we evaluate code LMs for their ability to comprehend both functional and non-functional aspects of code (§ 4.3) using NoFunClassify (§ 2.3) and HumanEvalClassify (§ 2.4). Prior works have considered code-comprehension tasks like Clone-detection (Svajlenko et al., 2014; Lu et al., 2021), Defect-detection (Zhou et al., 2019; Li et al., 2021; Lu et al., 2021), Code-explaination (Muennighoff et al., 2023; Leinonen et al., 2023), and Question Answering over Code (Huang et al., 2021; Liu & Wan, 2021; Lee et al., 2022; Sahu et al., 2024). Evaluating LMs on NoFunClassify and HumanEvalClassify allowed us to directly compare comprehension abilities of code LMs with their edit abilities (NoFunEdit and HumanEvalFix) in a controlled manner.

6. Limitations

Many LMs fail to follow the output formats specified in the prompts, resulting in the code not being extracted in a number of generations. We try to overcome this limitation by designing a parser that attempts to programmatically extract code when the expected output format is not followed. We use prompt templates which are intuitive to developers. However, the LMs are susceptible to small changes in the prompts. Further, due to limited test data and context size, we considered only 1-Shot prompts which could be extended to multi-shot in future.

Our metrics also have their own limitations. Due to the diversity of examples, runtime platforms and targeted non-functional scenarios, it was not possible to evaluate generations through execution for all requirements. We use the next best approximation (DiffBLEU) by checking similarity of diffs between source and generated code, against source and ground-truth code. For reported code runtimes, there are limited test cases and the numbers are benchmarked on a specific system configuration. All speedups obtained are with respect to these configurations and may not generalize. Static analyses defined in CodeQL may not cover all corner cases. However, we use the analyses packaged with the CodeQL tool which are implemented by domain experts.

Finally, a possible threat to validity of our results is that some of the input code (Android repositories for Latency and Resource Utilization tasks) in our benchmark might have been seen by the LMs during training. *However*, the LMs are unlikely to have seen the prompts constructed by us paired with the expected code revisions during training (note that we do not use the actual commit messages explicitly). Therefore, our results can be attributed to the ability (or lack thereof) of the LMs to follow the instructions, their knowledge of programming languages and the hints and domain knowledge we provide in our prompts.

7. Conclusions

Code LMs are assuming an important role in the art and craft of software engineering. While we are optimistic about their continued utility, we believe that they should be evaluated on scenarios that matter to practitioners. We focus on two of them in this paper: ability to improve code as per non-functional requirements and ability to comprehend the relation between requirements and code semantics. The former would broaden the scope of software engineering activities where the LMs can be useful and the latter will increase the trust of developers in LMs. Our extensive experiments found that the LMs falter on both these counts. With the rapid progress in the field, benchmarks can quickly become irrelevant. A remedy is to continuously improve the benchmarks. Our future goal is to keep extending NoFunEval by adding more languages, labeled examples spanning different requirements, and evaluation harnesses.

References

Agrawal, L. A., Kanade, A., Goyal, N., Lahiri, S. K., and Rajamani, S. K. Guiding language models of code with global context using monitors, 2023.

Al Madi, N. How readable is model-generated code? examining readability and visual inspection of github copilot. In *Proceedings of the 37th IEEE/ACM International Conference on Automated Software Engineering*, pp. 1–5, 2022.

Austin, J., Odena, A., Nye, M., Bosma, M., Michalewski, H., Dohan, D., Jiang, E., Cai, C., Terry, M., Le, Q., and Sutton, C. Program Synthesis with Large Language Models, August 2021. URL http://arxiv.org/abs/2108.07732. arXiv:2108.07732 [cs].

Bairi, R., Sonwane, A., Kanade, A., C, V. D., Iyer, A., Parthasarathy, S., Rajamani, S., Ashok, B., and Shet, S. Codeplan: Repository-level coding using llms and planning, 2023.

Ben Allal, L., Muennighoff, N., Kumar Umapathi, L., Lipkin, B., and von Werra, L. A framework for the evaluation of code generation models. https://github.com/bigcode-project/bigcode-evaluation-harness, 2022.

Bhatt, M., Chennabasappa, S., Nikolaidis, C., Wan, S., Evtimov, I., Gabi, D., Song, D., Ahmad, F., Aschermann, C., Fontana, L., Frolov, S., Giri, R. P., Kapil, D., Kozyrakis, Y., LeBlanc, D., Milazzo, J., Straumann, A., Synnaeve, G., Vontimitta, V., Whitman, S., and Saxe, J. Purple llama cyberseceval: A secure coding benchmark for language models, 2023.

Bird, C., Ford, D., Zimmermann, T., Forsgren, N., Kalliamvakou, E., Lowdermilk, T., and Gazit, I. Taking flight with copilot: Early insights and opportunities of ai-powered pair-programming tools. *Queue*, 20(6):35–57, 2022.

Black, S., Biderman, S., Hallahan, E., Anthony, Q., Gao, L., Golding, L., He, H., Leahy, C., McDonell, K., Phang, J., and others. Gpt-neox-20b: An open-source autoregressive language model. *arXiv preprint arXiv:2204.06745*, 2022.

Callan, J., Krauss, O., Petke, J., and Sarro, F. How do android developers improve non-functional properties of software? *Empir. Softw. Eng.*, 27(5):113, 2022. doi: 10.1007/s10664-022-10137-2. URL https://doi.org/10.1007/s10664-022-10137-2.

Cassano, F., Gouwar, J., Nguyen, D., Nguyen, S., Phipps-Costin, L., Pinckney, D., Yee, M.-H., Zi, Y., Anderson, C. J., Feldman, M. Q., Guha, A., Greenberg, M., and Jangda, A. Multipl-e: A scalable and polyglot approach to benchmarking neural code generation. *IEEE Transactions on Software Engineering*, 49(7):3675–3691, 2023. doi: 10.1109/TSE.2023.3267446.

Chen, M., Tworek, J., Jun, H., Yuan, Q., Pinto, H. P. d. O., Kaplan, J., Edwards, H., Burda, Y., Joseph, N., Brockman, G., and others. Evaluating large language models trained on code. *arXiv preprint arXiv:2107.03374*, 2021.

Chung, L., Nixon, B. A., Yu, E., and Mylopoulos, J. *Non-functional requirements in software engineering*, volume 5. Springer Science & Business Media, 2012.

CWE. The cwe top 25, 2021. URL https://cwe.mitre.org/top25/archive/2021/2021_cwe_top25.html#cwe_top_25.

Etemadi, K., Harrand, N., Larsén, S., Adzemovic, H., Phu, H. L., Verma, A., Madeiral, F., Wikström, D., and Monperrus, M. Sorald: Automatic patch suggestions for sonarqube static analysis violations. *IEEE Transactions on Dependable and Secure Computing*, 20(4):2794–2810, 2023. doi: 10.1109/TDSC.2022.3167316.

Fan, A., Lewis, M., and Dauphin, Y. Hierarchical neural story generation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pp. 889–898, 2018.

Fried, D., Aghajanyan, A., Lin, J., Wang, S., Wallace, E., Shi, F., Zhong, R., Yih, W.-t., Zettlemoyer, L., and Lewis, M. Incoder: A generative model for code infilling and synthesis. *arXiv preprint arXiv:2204.05999*, 2022.

Garg, S., Moghaddam, R. Z., Clement, C. B., Sundaresan, N., and Wu, C. Deepperf: A deep learning-based approach for improving software performance. *arXiv preprint arXiv:2206.13619*, 2022.

Guo, D., Zhu, Q., Yang, D., Xie, Z., Dong, K., Zhang, W., Chen, G., Bi, X., Wu, Y., Li, Y. K., Luo, F., Xiong, Y., and Liang, W. Deepseek-coder: When the large language model meets programming – the rise of code intelligence, 2024.

Gupta, R., Pal, S., Kanade, A., and Shevade, S. Deepfix: fixing common c language errors by deep learning. In *Proceedings of the Thirty-First AAAI Conference on Artificial Intelligence*, pp. 1345–1351, 2017.

He, J. and Vechev, M. Large language models for code: Security hardening and adversarial testing. In *Proceedings of the 2023 ACM SIGSAC Conference on Computer and Communications Security*, pp. 1865–1879, 2023.

Hendrycks, D., Basart, S., Kadavath, S., Mazeika, M., Arora, A., Guo, E., Burns, C., Puranik, S., He, H., Song, D., and

Steinhardt, J. Measuring coding challenge competence with apps. In Vanschoren, J. and Yeung, S. (eds.), *Proceedings of the Neural Information Processing Systems Track on Datasets and Benchmarks*, volume 1. Curran, 2021.

Hoffmann, J., Borgeaud, S., Mensch, A., Buchatskaya, E., Cai, T., Rutherford, E., Casas, D. d. L., Hendricks, L. A., Welbl, J., Clark, A., et al. Training compute-optimal large language models. *arXiv preprint arXiv:2203.15556*, 2022.

Holtzman, A., Buys, J., Du, L., Forbes, M., and Choi, Y. The curious case of neural text degeneration. In *International Conference on Learning Representations*, 2019.

Huang, J., Tang, D., Shou, L., Gong, M., Xu, K., Jiang, D., Zhou, M., and Duan, N. Cosqa: 20, 000+ web queries for code search and question answering. In *Proceedings of the 59th Annual Meeting of the Association for Computational Linguistics and the 11th International Joint Conference on Natural Language Processing, ACL/IJCNLP*. Association for Computational Linguistics, 2021.

Jain, N., Zhang, T., Chiang, W.-L., Gonzalez, J. E., Sen, K., and Stoica, I. Llm-assisted code cleaning for training accurate code generators, 2023.

Jiang, A. Q., Sablayrolles, A., Mensch, A., Bamford, C., Chaplot, D. S., Casas, D. d. l., Bressand, F., Lengyel, G., Lample, G., Saulnier, L., et al. Mistral 7b. *arXiv preprint arXiv:2310.06825*, 2023.

Kaplan, J., McCandlish, S., Henighan, T., Brown, T. B., Chess, B., Child, R., Gray, S., Radford, A., Wu, J., and Amodei, D. Scaling laws for neural language models. *arXiv preprint arXiv:2001.08361*, 2020.

Kwon, W., Li, Z., Zhuang, S., Sheng, Y., Zheng, L., Yu, C. H., Gonzalez, J. E., Zhang, H., and Stoica, I. Efficient memory management for large language model serving with pagedattention. *arXiv preprint arXiv:2309.06180*, 2023.

Lai, Y., Li, C., Wang, Y., Zhang, T., Zhong, R., Zettlemoyer, L., tau Yih, S. W., Fried, D., Wang, S., and Yu, T. Ds-1000: A natural and reliable benchmark for data science code generation, 2022.

Landes, D. and Studer, R. The treatment of non-functional requirements in mike. In *European software engineering conference*, pp. 294–306. Springer, 1995.

Lee, C., Seonwoo, Y., and Oh, A. CS1QA: A dataset for assisting code-based question answering in an introductory programming course. In *Proceedings of the 2022 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies*, pp. 2026–2040, 2022.

Leinonen, J., Denny, P., MacNeil, S., Sarsa, S., Bernstein, S., Kim, J., Tran, A., and Hellas, A. Comparing code explanations created by students and large language models. *arXiv preprint arXiv:2304.03938*, 2023.

Li, J., Hui, B., Qu, G., Li, B., Yang, J., Li, B., Wang, B., Qin, B., Cao, R., Geng, R., et al. Can llm already serve as a database interface? a big bench for large-scale database grounded text-to-sqls. *arXiv preprint arXiv:2305.03111*, 2023a.

Li, R., Allal, L. B., Zi, Y., Muennighoff, N., Kocetkov, D., Mou, C., Marone, M., Akiki, C., Li, J., Chim, J., et al. Starcoder: may the source be with you! *arXiv preprint arXiv:2305.06161*, 2023b.

Li, Y., Choi, D., Chung, J., Kushman, N., Schrittwieser, J., Leblond, R., Eccles, T., Keeling, J., Gimeno, F., Lago, A. D., Hubert, T., Choy, P., d'Autume, C. d. M., Babuschkin, I., Chen, X., Huang, P.-S., Welbl, J., Gowal, S., Cherepanov, A., Molloy, J., Mankowitz, D. J., Robson, E. S., Kohli, P., Freitas, N. d., Kavukcuoglu, K., and Vinyals, O. Competition-level code generation with AlphaCode. *Science*, 378(6624): 1092–1097, 2022. doi: 10.1126/science.abq1158. URL `https://www.science.org/doi/abs/10.1126/science.abq1158`. _eprint: https://www.science.org/doi/pdf/10.1126/science.abq1158.

Li, Z., Zou, D., Xu, S., Jin, H., Zhu, Y., and Chen, Z. Sysevr: A framework for using deep learning to detect software vulnerabilities. *IEEE Transactions on Dependable and Secure Computing*, 19(4):2244–2258, 2021.

Liu, C. and Wan, X. Codeqa: A question answering dataset for source code comprehension. In *Findings of the Association for Computational Linguistics: EMNLP*. Association for Computational Linguistics, 2021.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023a.

Liu, J., Xia, C. S., Wang, Y., and Zhang, L. Is your code generated by chatgpt really correct? rigorous evaluation of large language models for code generation. *arXiv preprint arXiv:2305.01210*, 2023b.

Loriot, B., Madeiral, F., and Monperrus, M. Styler: learning formatting conventions to repair checkstyle violations. *Empirical Software Engineering*, 27(6), aug 2022. doi: 10.1007/s10664-021-10107-0. URL `https://doi.org/10.1007%2Fs10664-021-10107-0`.

Lu, S., Guo, D., Ren, S., Huang, J., Svyatkovskiy, A., Blanco, A., Clement, C., Drain, D., Jiang, D., Tang, D., et al. Codexglue: A machine learning benchmark dataset for code understanding and generation. *arXiv preprint arXiv:2102.04664*, 2021.

Luo, Z., Xu, C., Zhao, P., Sun, Q., Geng, X., Hu, W., Tao, C., Ma, J., Lin, Q., and Jiang, D. Wizardcoder: Empowering code large language models with evol-instruct. *arXiv preprint arXiv:2306.08568*, 2023.

Madaan, A., Shypula, A., Alon, U., Hashemi, M., Ranganathan, P., Yang, Y., Neubig, G., and Yazdanbakhsh, A. Learning performance-improving code edits. *arXiv preprint arXiv:2302.07867*, 2023.

Moura, I., Pinto, G., Ebert, F., and Castor, F. Mining energy-aware commits. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*, pp. 56–67, 2015. doi: 10.1109/MSR.2015.13.

Muennighoff, N., Liu, Q., Zebaze, A., Zheng, Q., Hui, B., Zhuo, T. Y., Singh, S., Tang, X., von Werra, L., and Longpre, S. Octopack: Instruction tuning code large language models, 2023.

Nijkamp, E., Hayashi, H., Xiong, C., Savarese, S., and Zhou, Y. Codegen2: Lessons for training llms on programming and natural languages. *arXiv preprint arXiv:2305.02309*, 2023.

OpenAI. Gpt-4 technical report, 2023.

Ouyang, L., Wu, J., Jiang, X., Almeida, D., Wainwright, C. L., Mishkin, P., Zhang, C., Agarwal, S., Slama, K., Ray, A., Schulman, J., Hilton, J., Kelton, F., Miller, L., Simens, M., Askell, A., Welinder, P., Christiano, P., Leike, J., and Lowe, R. Training language models to follow instructions with human feedback, 2022.

Pearce, H., Ahmad, B., Tan, B., Dolan-Gavitt, B., and Karri, R. Asleep at the keyboard? assessing the security of github copilot's code contributions. In *2022 IEEE Symposium on Security and Privacy (SP)*, pp. 754–768. IEEE, 2022.

Perry, N., Srivastava, M., Kumar, D., and Boneh, D. Do users write more insecure code with ai assistants? *arXiv preprint arXiv:2211.03622*, 2022.

Puri, R., Kung, D. S., Janssen, G., Zhang, W., Domeniconi, G., Zolotov, V., Dolby, J., Chen, J., Choudhury, M., Decker, L., et al. Project codenet: A large-scale ai for code dataset for learning a diversity of coding tasks. *arXiv preprint arXiv:2105.12655*, 1035, 2021.

Raychev, V., Bielik, P., and Vechev, M. T. Probabilistic model for code with decision trees. *Proceedings of the 2016 ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2016. URL https://api.semanticscholar.org/CorpusID:2658344.

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

Sahu, S. P., Mandal, M., Bharadwaj, S., Kanade, A., Maniatis, P., and Shevade, S. CodeQueries: A Dataset of Semantic Queries over Code. In *17th Innovations in Software Engineering Conference*. ACM, 2024.

Svajlenko, J., Islam, J. F., Keivanloo, I., Roy, C. K., and Mia, M. M. Towards a big data curated benchmark of inter-project code clones. In *2014 IEEE International Conference on Software Maintenance and Evolution*, pp. 476–480. IEEE, 2014.

Taori, R., Gulrajani, I., Zhang, T., Dubois, Y., Li, X., Guestrin, C., Liang, P., and Hashimoto, T. B. Stanford alpaca: An instruction-following llama model. https://github.com/tatsu-lab/stanford_alpaca, 2023.

Tony, C., Mutas, M., Díaz Ferreyra, N., and Scandariato, R. Llmseceval: A dataset of natural language prompts for security evaluations. In *2023 IEEE/ACM 20th International Conference on Mining Software Repositories (MSR)*, 2023. doi: 10.5281/zenodo.7565965.

Tunstall, L., Von Werra, L., and Wolf, T. *Natural language processing with transformers*. " O'Reilly Media, Inc.", 2022.

Wadhwa, N., Pradhan, J., Sonwane, A., Sahu, S. P., Natarajan, N., Kanade, A., Parthasarathy, S., and Rajamani, S. Frustrated with code quality issues? llms can help!, 2023.

Wang, B. and Komatsuzaki, A. GPT-J-6B: A 6 billion parameter autoregressive language model, 2021.

Wang, Y., Le, H., Gotmare, A. D., Bui, N. D., Li, J., and Hoi, S. C. Codet5+: Open code large language models for code understanding and generation. *arXiv preprint arXiv:2305.07922*, 2023.

Wei, J., Bosma, M., Zhao, V. Y., Guu, K., Yu, A. W., Lester, B., Du, N., Dai, A. M., and Le, Q. V. Finetuned language models are zero-shot learners. *arXiv preprint arXiv:2109.01652*, 2021.

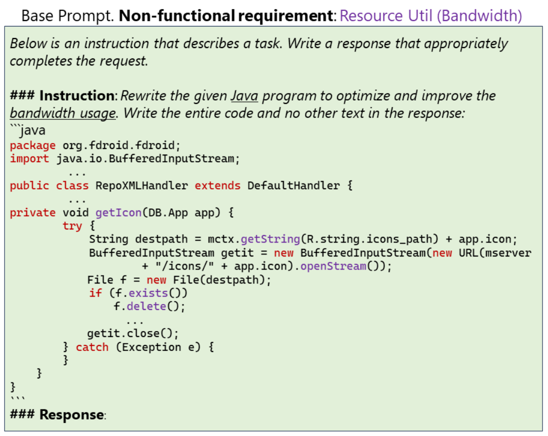
Wei, J., Wang, X., Schuurmans, D., Bosma, M., Xia, F., Chi, E., Le, Q. V., Zhou, D., et al. Chain-of-thought prompting elicits reasoning in large language models. *Advances in Neural Information Processing Systems*, 35: 24824–24837, 2022.

West, P., Lu, X., Dziri, N., Brahman, F., Li, L., Hwang, J. D., Jiang, L., Fisher, J., Ravichander, A., Chandu, K., et al. The generative ai paradox:" what it can create, it may not understand". *arXiv preprint arXiv:2311.00059*, 2023.

Yu, T., Zhang, R., Yang, K., Yasunaga, M., Wang, D., Li, Z., Ma, J., Li, I., Yao, Q., Roman, S., Zhang, Z., and Radev, D. Spider: A large-scale human-labeled dataset for complex and cross-domain semantic parsing and text-to-sql task. In *Proceedings of the 2018 Conference on Empirical Methods in Natural Language Processing*, Brussels, Belgium, 2018. Association for Computational Linguistics.

Zhou, Y., Liu, S., Siow, J., Du, X., and Liu, Y. Devign: Effective vulnerability identification by learning comprehensive program semantics via graph neural networks. *Advances in neural information processing systems*, 32, 2019.

A. Appendix

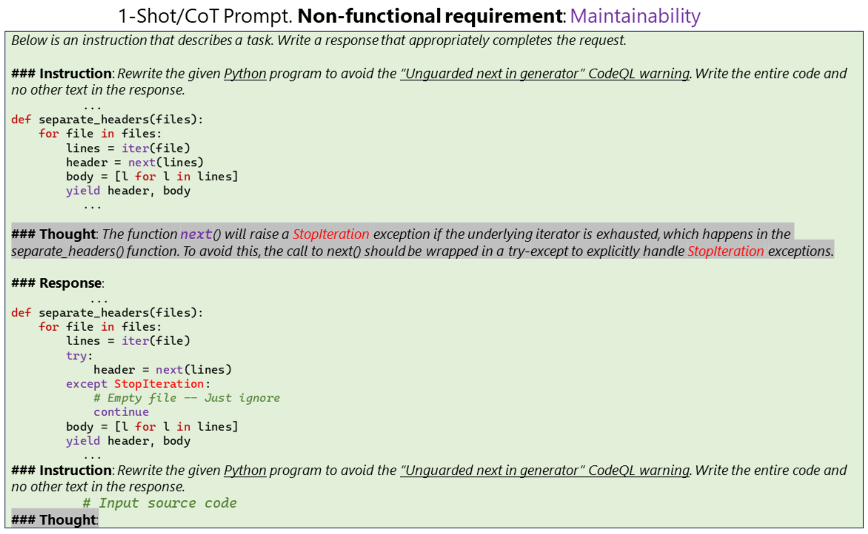


Base Prompt. **Non-functional requirement**: Resource Util (Bandwidth)

~~~~
Below is an instruction that describes a task. Write a response that appropriately
completes the request.

### Instruction: Rewrite the given Java program to optimize and improve the
bandwidth usage. Write the entire code and no other text in the response:
```java
package org.fdroid.fdroid;
import java.io.BufferedInputStream;
        ...
public class RepoXMLHandler extends DefaultHandler {
        ...
private void getIcon(DB.App app) {
        try {
            String destpath = mctx.getString(R.string.icons_path) + app.icon;
            BufferedInputStream getit = new BufferedInputStream(new URL(mserver
                    + "/icons/" + app.icon).openStream());
            File f = new File(destpath);
            if (f.exists())
                f.delete();
                  ...
            getit.close();
        } catch (Exception e) {
        }
    }
}
```
### Response:
~~~~

Figure 8: An example Base Prompt for improving bandwidth usage in code for an android application (§ 2.2).



1-Shot/CoT Prompt. **Non-functional requirement**: Maintainability

~~~~
Below is an instruction that describes a task. Write a response that appropriately completes the request.

### Instruction: Rewrite the given Python program to avoid the "Unguarded next in generator" CodeQL warning. Write the entire code and
no other text in the response.
          ...
def separate_headers(files):
    for file in files:
        lines = iter(file)
        header = next(lines)
        body = [l for l in lines]
        yield header, body
          ...

### Thought: The function next() will raise a StopIteration exception if the underlying iterator is exhausted, which happens in the
separate_headers() function. To avoid this, the call to next() should be wrapped in a try-except to explicitly handle StopIteration exceptions.

### Response:
          ...
def separate_headers(files):
    for file in files:
        lines = iter(file)
        try:
            header = next(lines)
        except StopIteration:
            # Empty file -- Just ignore
            continue
        body = [l for l in lines]
        yield header, body
          ...
### Instruction: Rewrite the given Python program to avoid the "Unguarded next in generator" CodeQL warning. Write the entire code and
no other text in the response.
          # Input source code
### Thought:
~~~~

Figure 9: An example 1-Shot / Chain-of-Thought prompt template for fixing a maintainability issue ("Unguarded next in generator") as flagged by CodeQL. The underlined texts are instantiated based on the example. The shaded text denotes the reasoning we include for the corresponding Chain-of-Thought prompt (§ 2.2).

A.1. Prompt Templates

Figure 8 provides an example base prompt used for prompting models with the high-level non-functional requirement to be achieved in editing the code. Figure 9 highlights the template used for prompting models with a single example along with a thought relevant for the non-functional requirement being addressed (§ 2.2).

A.2. Additional Results

Figure 10 provides a summary of all the absolute values of the evaluation numbers obtained across models and prompts. The darker shades indicating better performance on that specific task (§ 4.2).

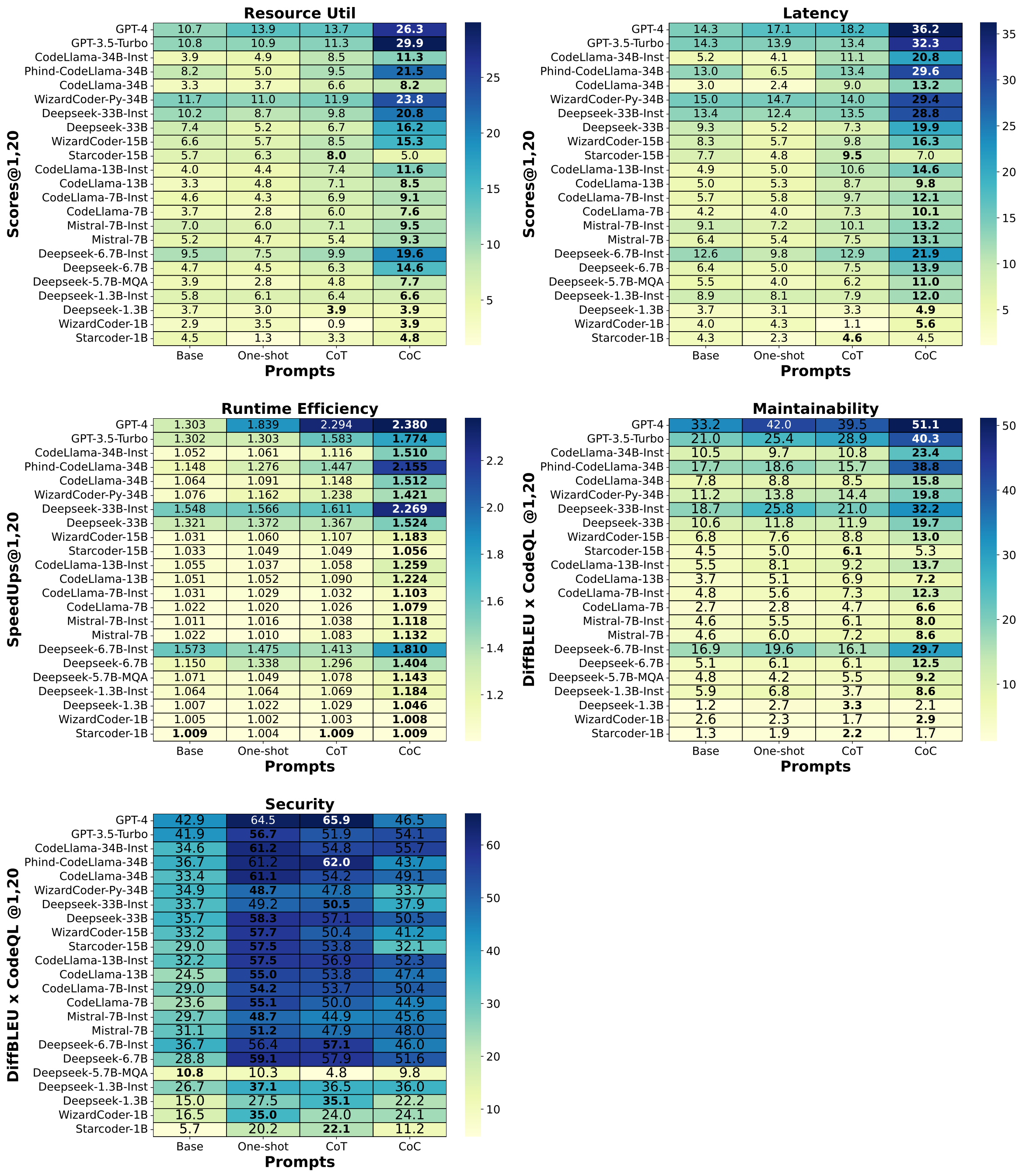


Figure 10: Performance of code LMs on the NoFunEdit dataset by different non-functional requirements, for the four prompts (§ 4.2).